

Spring 4-27-2012

Design and Implementation of Fault Tolerant Adders on Field Programmable Gate Arrays

Lakshmi Phani Deepthi Bollepalli

Follow this and additional works at: https://scholarworks.uttyler.edu/ee_grad



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Bollepalli, Lakshmi Phani Deepthi, "Design and Implementation of Fault Tolerant Adders on Field Programmable Gate Arrays" (2012). *Electrical Engineering Theses*. Paper 17.
<http://hdl.handle.net/10950/62>

This Thesis is brought to you for free and open access by the Electrical Engineering at Scholar Works at UT Tyler. It has been accepted for inclusion in Electrical Engineering Theses by an authorized administrator of Scholar Works at UT Tyler. For more information, please contact tbianchi@uttyler.edu.

**DESIGN AND IMPLEMENTATION OF FAULT TOLERANT
ADDERS ON FIELD PROGRAMMABLE GATE ARRAYS**

by

Lakshmi Phani Deepthi Bollepalli

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Electrical Engineering
Department of Electrical Engineering

David H. K. Hoe, Ph.D., Committee Chair

College of Engineering and Computer Science

The University of Texas at Tyler
May 2012

The University of Texas at Tyler
Tyler, Texas

This is to certify that the Master's thesis of

Lakshmi Phani Deepthi Bollepalli

has been approved for the thesis requirements on
March 22nd, 2012
for the Master of Science Degree in Electrical Engineering

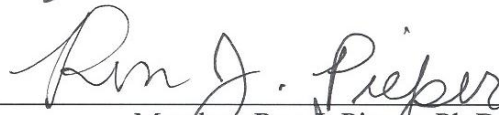
Approvals:



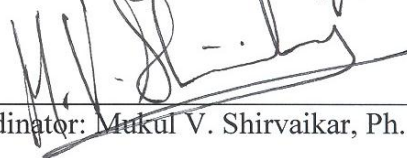
Thesis Chair: David H. K. Hoe, Ph.D.



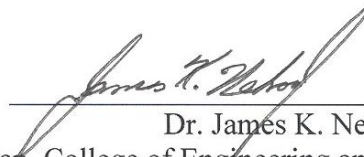
Member: Mukul V. Shirvaikar, Ph. D.



Member: Ron J. Pieper, Ph.D.



Chair and Graduate Coordinator: Mukul V. Shirvaikar, Ph. D.



Dr. James K. Nelson, Jr., Ph.D., P.E.,
Dean, College of Engineering and Computer Science,
Brazzel Professor of Engineering

Acknowledgements

Firstly, I sincerely thank my god for bestowing his divine blessings on me in successfully accomplishing this task. Secondly, my hearty thanks to my family members: my grandfather Purna Chandra Rao Ravi, mom Vijaya Kumari, uncle Rama Krishna Prasad Ravi and my loving sisters Gayathri and Hanumasri for their wholehearted support, love and encouragement for making my dream come true. I would like to express my honest and heartfelt gratitude to my advisor Dr. David Hoe for his encouragement, patience, supervision and constant support from the preliminary stages to the concluding level on me without whom this thesis would not have been possible. Also, I am grateful to my professor Dr. Hoe for encouraging me with the saying, “Give it the extra push.” Without his support and encouragement from the very first day of my work, I cannot imagine my successful completion of this thesis. I am very grateful to my friend Chris Martinez for spending his valuable time throughout my research, working on late nights and weekends in the lab by encouraging me in completion of this task in every aspect. Also I would like to thank my seniors Venkata Chandra Sekhar Mandala and Rahul Jesuran and for making me active and supporting me throughout my Master’s.

I would like to thank my committee members, Dr. Ron J. Piper and Dr. Mukul V. Shirvaikar for taking time and for reviewing my work. I still remember the precious words by Dr. Shirvaikar on the way home regarding my research with Dr. David Hoe which gave a million tons of encouragement and will power for a successful start. I would like to express my profound gratitude to him for his constant support and guidance throughout my Master’s program. Finally I would like to thank the entire EE department and the University of Texas at Tyler for supporting me throughout my Master’s. Finally, I would like to thank all those who supported me in any respect during the completion of the thesis.

Table of Contents

Chapter One	1
Introduction.....	1
1.1 Importance of Fault Tolerance in FPGAs	2
1.2 Review of the Relevant Literature:	2
1.3 Research Objectives	3
1.4 Research Method	3
1.5 Thesis Outline.....	4
Chapter Two.....	5
Fault Tolerance on FPGAs.....	5
2.1 Introduction	5
2.2 Basic Adder Designs	5
2.2.1 Full Adder.....	5
2.3 Ripple Carry Adder	7
2.4 Kogge-Stone Adder.....	7
2.4.1 8-bit Kogge-Stone Adder.....	8
2.5 Sparse Kogge-Stone Adder	13
2.6. Basic Fault Tolerance - Hardware Redundancy	14
2.7 Advanced Fault Tolerant Methods	15
2.7.1 Structural Design – Hybrid Approach	15
2.7.2 Roving	17
2.7.3 Graceful Degradation	19
Summary	21
Chapter 3.....	22
Basic Fault Tolerant Implementation.....	22
3.1 Introduction	22
3.2 FPGA Implementation Method	22
3.3 Triple Modular Redundancy-RCA	22
3.4 Regular Kogge-Stone Adder Fault Correction Approach	24
3.5 Lower Half Fault Tolerant Sparse Kogge-Stone Adder	25

3.5.1 Simulations of the Sparse Kogge-Stone Adder	27
Summary	28
Chapter 4.....	29
Advanced Fault Tolerance Concepts	29
4.1 Introduction	29
4.2 Upper Half Fault Tolerant Sparse Kogge-Stone Adder	29
4.2.1 Simulation Results	33
4.3 Graceful Degradation	35
4.3.1 Implementation	35
4.3.2 Simulation Results	36
4.4 Synthesis Results	38
4.5 Hardware Implementation	41
Summary	44
Chapter Five.....	46
Conclusions and Future Work	46
5.1 Conclusions	46
5.2 Future Work	46
References.....	48
Appendices.....	50
Appendix: A.....	51
A1. VHDL Code for 32-bit TMR-RCA	51
A2. VHDL Code for adder1 in 32-bit TMR-RCA	52
A3. VHDL Code for adder2 in 32-bit TMR-RCA	53
A4. VHDL Code for adder3 in 32-bit TMR-RCA	54
A5. VHDL Code for comparator in 32-bit TMR-RCA.....	54
Appendix: B	56
B1. VHDL Code for 8-bit Kogge-Stone Fault Correcting Adder	56
B2. VHDL Code for mux in 8-bit Kogge-Stone Fault Correcting Adder	62
B3. VHDL Code for GPblock in 8-bit Kogge-Stone Fault Correcting Adder	62
B4. VHDL Code for blackcell in 8-bit Kogge-Stone Fault Correcting Adder.....	63
B5. VHDL Code for graycell in 8-bit Kogge-Stone Fault Correcting Adder	63

B6. VHDL Code for faultgraycell in 8-bit Kogge-Stone Fault Correcting Adder....	64
B7. VHDL Code for buffer1 in 8-bit Kogge-Stone Fault Correcting Adder	64
B8. VHDL Code for outmux in 8-bit Kogge-Stone Fault Correcting Adder.....	65
B9. VHDL Code for sum in 8-bit Kogge-Stone Fault Correcting Adder	65
B10. VHDL Code for CntlMuxs in 8-bit Kogge-Stone Fault Correcting Adder	66
Appendix: C	68
C1. VHDL Code for 32-bit Kogge-Stone Adder (Lower half)	68
C2. VHDL Code for ConcatenationRCA in 32-bit Kogge-Stone Adder (Lower half)	77
C3. VHDL Code for FaultyAdder1 in 32-bit Kogge-Stone Adder (Lower half).....	78
C4. VHDL Code for comparator1 in 32-bit Kogge-Stone Adder (Lower half).....	78
C5. VHDL Code for bitcounter1 in 32-bit Kogge-Stone Adder (Lower half).....	80
Appendix: D.....	81
D1. VHDL Code for 32-bit Kogge-Stone Adder (Upper half)	81
D2. VHDL Code for greengroup in 32-bit Kogge-Stone Adder (Upper half)	85
D3. VHDL Code for purplegroup in 32-bit Kogge-Stone Adder (Upper half).....	87
D4. VHDL Code for bluegroup in 32-bit Kogge-Stone Adder (Upper half)	89
D5. VHDL Code for ConcatenationRCA in 32-bit Kogge-Stone Adder (Upper half)	91
D6. VHDL Code for faultgraycell in 32-bit Kogge-Stone Adder (Upper half)	92
D7. VHDL Code for faultblackcell in 32-bit Kogge-Stone Adder (Upper half)	92
Appendix: E	94
E1. VHDL Code for 32-bit Graceful Degradation	94
E2. VHDL Code for comparator2 in 32-bit Graceful Degradation.....	104
Appendix: F	106
F1. VHDL code for 32-bit TMR-RCA Implemented on Hardware	106
F2. VHDL code for 32-bit Sparse Kogge-Stone Lower Half Implemented on Hardware.....	108
F3. VHDL code for 32-bit Graceful Degradation Implemented on Hardware	110
Appendix G.....	113
G.1 Other Fault Combinations for Upper Half Fault Tolerant Sparse Kogge-Stone Adder.....	113

Appendix H.....	115
H1 Spartan-3E	115
H2 Virtex-5.....	116
Appendix I	118
I1. Delay Calculation of TMR_RCA on Logic Analyzer.....	118
I2. Observing Worst Case Transition for Kogge Stone Adder.....	124
Appendix J	133

List of Figures

Figure 2.1 Block diagram of a full adder	6
Figure 2.2 4-bit ripple carry adder	7
Figure 2.3 Generate-Propagate block	9
Figure 2.4(a) Black cell	9
Figure 2.4(a) Gray cell	9
Figure 2.5 Buffer.....	10
Figure 2.6 8-bit Kogge-Stone adder.....	11
Figure 2.7 16-bit Kogge-Stone adder.....	12
Figure 2.8 Sparse Kogge-Stone adder	13
Figure 2.9 General Triple Modular Redundancy	14
Figure 2.10 TMR adder circuit using ripple carry	15
Figure 2.11 8-bit Kogge-Stone carry tree illustrating the mutually exclusive even and odd carry trees	16
Figure 2.12 Timing diagram for three adders in execution unit (T_c is the clock period) ..	17
Figure 2.13 Block diagram for the proposed 8-bit fault tolerant Kogge-Stone adder	18
Figure 2.14 Roving area under test across the chip	18
Figure 2.15 General block diagram of graceful degradation	19
Figure 2.16 General view of the graceful degradation process	20
Figure 3.1 General design flow.....	23
Figure 3.2 Simulation results for the 64-bit TMR-RCA.....	24
Figure 3.3 Simulation results for the 64-bit error correcting Kogge-Stone adder	25
Figure 3.4 Block diagram of fault tolerant sparse Kogge-Stone adder.....	26
Figure 3.5 Timing diagram for the lower half fault tolerant Kogge-Stone.....	27
Figure 3.6 Simulation results for the 64-bit lower half FT sparse Kogge-Stone adder	28

Figure 4.1 Upper half detection scheme for 16-bit sparse Kogge-Stone adder	30
Figure 4.2 Upper half detection truth table	30
Figure 4.3 Upper half detection scheme with fault free carry comparisons	31
Figure 4.4 Truth table for upper half error detection with fault free comparisons	32
Figure 4.5 Upper half detection scheme for a 32-bit sparse Kogge-Stone adder	33
Figure 4.6 (a) Normal adder operation of the sparse Kogge-Stone upper half	34
Figure 4.6 (b) Fault tolerant adder operation of the sparse Kogge-Stone upper half	34
Figure 4.7 Block diagram of graceful degradation approach on sparse Kogge-Stone adder	35
Figure 4.8 (a) Normal adder operation of the graceful degradation adder	37
Figure 4.8 (b) Fault tolerant adder operation of the graceful degradation adder	37
Figure 4.9 Estimation of resources used from FPGA synthesis	38
Figure 4.10 Corresponding delays for the sparse KS on Sparatn 3E FPGA.....	39
Figure 4.11 Delay of FT adders on Spartan 3E FPGA	40
Figure 4.12 Delay of FT adders on Virtex 5 FPGA.....	41
Figure 4.13 Measured delay for the 64-bit TMR-RCA	42
Figure 4.14: Implemented procedure for simulating the worst-case delay on the sparse Kogge-Stone lower half approach	43
Figure 4.15 Measured delay for the 64-bit sparse Kogge-Stone Adder	43
Figure 4.16 Measured delay for the 64-bit graceful degradation adder.....	44
Figure 4.17 Summary of adder delays on Spartan 3E	45
Figure G: Fault tolerant adder operation for multiple fault combinations.....	114
Figure H-1: Spartan-3E FPGA.....	115
Figure H-2: Virtex-5 FPGA	116
Figure I-1: Adder delay including ROM and multiplexer	118
Figure I-2: Logic cell of Spartan 3E	120

Figure I-3 Adder delay excluding ROM and multiplexer.....	123
Figure I-4: Kogge-Stone adder with selected BC1	126
Figure I-5: Kogge-Stone adder with selected GC0.....	127
Figure I.6: Kogge-Stone adder with selected GC2	128

List of Tables

Table 2.1: Truth table of full adder	6
Table 2.6: Kogge-Stone adders of different bit-widths	11
Table I-1: Input pattern chosen for testing TMR-RCA	119
Table I-2: Worst case carry and sum delays	121
Table I-3: Worst case input delays.....	123
Table I-4: Subset of (g, p) relations used for testing.....	125
Table I-5: Black cell 1 outputs for high inputs	126
Table I-6: Gray cell 0 outputs for high inputs.....	127
Table I-7: Gray cell 2 outputs for high inputs.....	128
Table I-8: Black cell 1 outputs for one low input	129
Table I-9: Gray cell 0 outputs for one low input	130
Table I-10: Gray cell 2 outputs for one low input	131
Table J-1: Delay summary for lower half fault tolerant sparse Kogge-Stone adder	133

Abstract

DESIGN AND IMPLEMENTATION OF FAULT TOLERANT ADDERS ON FIELD PROGRAMMABLE GATE ARRAYS

Lakshmi Phani Deepthi Bollepalli

Thesis chair: David H. K. Hoe, Ph.D.

The University of Texas at Tyler

May 2012

Fault tolerant systems play a very prominent role in many digital systems especially for those implemented with nanoscale technologies because of their susceptibility to electromagnetic interference and transient errors due to cosmic rays. Arithmetic logic circuits play a vital role in all digital signal processing systems and also in microprocessors. Keeping this in mind, this research is concerned with achieving fault tolerance on various adder architectures on Field Programmable Gate Arrays (FPGAs).

The research method involves implementing error detection and correction techniques for the sparse Kogge-Stone adder and comparing it with Triple Modular Redundancy (TMR) techniques. Fault tolerance is implemented on a Kogge-Stone adder by taking the advantage of inherent redundancy in the carry tree. On a sparse Kogge-Stone adder, fault tolerance is implemented by introducing additional ripple carry adders into the design. Implementing this fault tolerance approach on the sparse Kogge-Stone adder is successfully completed and verified by introducing faults either on the ripple carry adder or in the carry tree. The adder designs are specified using a high-level descriptor language called “Very High Speed Integrated Circuit Hardware Description Language” (VHDL) and implemented on an FPGA. Two types of Xilinx FPGAs were

used in this study: the Spartan 3E and Virtex 5. The fault tolerant adders were analyzed in terms of their delay and resource utilization as a function of their bit-widths.

The results of this research provide important design guidelines for the implementation of fault tolerant adders on FPGAs. The Triple Modular Redundancy-Ripple Carry Adder (TMR-RCA) is the most efficient approach for fault tolerant design on an FPGA in terms of its resources, due to its simplicity and the ability to take advantage of the fast-carry chain. However, for very large bit widths, there are indications that the sparse Kogge-Stone adder offers superior performance over an RCA when implemented on an FPGA. Two fault tolerant approaches were implemented using a sparse Kogge-Stone architecture. First, a fault tolerant sparse Kogge-Stone adder is designed by taking advantage of the existing ripple carry adders in the architecture and adopting a similar approach to the TMR-RCA by inserting two additional ripple carry adders into the design. Second, a graceful degradation approach is implemented with the sparse Kogge-Stone adder. In this approach, a faulty block is permanently replaced with a spare block. As the spare block is initially used for fault checking, the fault tolerant capability of the circuit is degraded in order to continue fault-free operation. The adder delay is smaller for graceful degradation by approximately 1 ns from measured results and 2 ns from the synthesis results independent of the bit widths when compared with the fault tolerant Kogge-Stone adder. However, the resource utilization is similar for both adders.

Chapter One

Introduction

Fault tolerance plays a very important role in modern systems where immediate human intervention is not possible and system failure can have disastrous consequences. A fault tolerant system has the ability to detect and then correct the occurrence of a hardware failure. In order to detect the fault, the system must be able to sense any deviations from its normal operation. A fully fault tolerant system also has the ability to correct the fault in order to return the system to its normal functionality. An optimal design will minimize the amount of extra logic required to detect and then correct the occurrence of the fault. An extreme temperature change is one of the reasons in which fault tolerance is necessary for devices operating in harsh operating environments, as found, for example, in space and military applications. Fault tolerance will also be necessary in nanoelectronic systems, as small device dimensions make the system more susceptible to outside interference, such as cosmic radiation.

This thesis will study methods to implement fault tolerant arithmetic circuits on Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits (ICs) that can be configured to implement a specific function after the chip has been manufactured. The first FPGAs were put on the market by Xilinx Corporation in 1985 and by the late 1990's FPGAs were becoming more popular than Application Specific Integrated Circuits (ASICs). The advantages of using FPGAs are their reprogrammable nature, ease of prototyping, rapid time to market, and minimal non-recurring engineering (NRE) cost compared to custom IC designs. Most modern electronic systems contain some high performance digital chips known as Digital Signal Processors (DSPs). DSP designs are commonly used in electronic systems such as avionics, communication systems and also in portable electronics. DSP chips transform and manipulate digitally encoded signals according to some specified system design goal. Various algorithms such as the Fast Fourier Transforms are used to analyze the signals which are in digital form. The main components of a DSP chip are the adder and multiplier along with memory elements. The

performance of the system is based on the speed at which arithmetic operations are performed. Hence the adder plays a vital role in DSPs for carrying out all the required arithmetic operations.

1.1 Importance of Fault Tolerance in FPGAs

A large portion of an FPGA chip consists of its configuration memory. The logic stored in the memory of the system can be altered by Single-Event Upsets (SEUs). SEUs can occur due to cosmic radiation or a high energy neutron striking the substrate of the device silicon. As SEUs can cause single-bit errors within the configuration memory, a fault tolerant system that uses FPGAs must guard against these occurrences. Fault tolerant systems are also important for circuits implemented on nanoscale technologies as external influences such as electromagnetic interference and cosmic rays can cause transient errors which in turn affect the operation of the devices and can degrade the system reliability. State-of-the-art FPGAs are designed with very fine geometrics making them susceptible to faults due to such electrical interference. For example, Xilinx's Virtex-6 FPGA uses 40 nm technology and the Virtex-7 uses 28 nm technology. The silicon processing technology is at the 22 nm mode at the time this thesis was written. Degradation mechanisms of the devices become more severe with the shrinking of the process geometry. For example, hot-carrier effects due to the increasing electric field strength in the transistor's channel causes a gradual degradation in the device performance through threshold voltage shifts.

1.2 Review of the Relevant Literature:

Previous studies have investigated fault tolerant methods implemented on FPGAs [1]. The basic fault tolerant approach is Triple Modular Redundancy (TMR) which is used as a point of reference to compare with advanced fault approach considered in this thesis [2]. TMR is a common solution for hardening digital logic against SEUs and is widely adopted in ASIC designs [3]. Hardware is essentially replicated in triplicate with a voter circuit used to pass the majority rule signals to the output.

Roving fault detection and graceful degradation are some of the fault tolerant approaches used for ensuring reliable FPGA designs [4]. Roving fault detection performs a progressive scan of an FPGA structure by swapping blocks with the same functionality

with a block carrying out the test function. Graceful degradation is an approach in which the faulty block is replaced with a spare block. A fault correction approach for a parallel prefix, N bit Kogge-Stone adder, which consists of two independent $N/2$ bit Han-Carlson (HC) adders, is implemented [5]. In addition, fault tolerance can be implemented by using self-testing areas (STARs) on an FPGA, which allows fault checking to occur without disturbing the normal system operation [6].

1.3 Research Objectives

As the adder plays a vital role in digital signal processing systems and microprocessors, the main objective of this research is to design and implement fault tolerant adders on FPGAs. Existing fault tolerant approaches will be applied to adders of varying bit widths for implementation on FPGAs.

1.4 Research Method

To meet the research objectives three adder topologies, namely the ripple carry adder, Kogge-Stone adder, and the sparse Kogge-Stone adder are studied. The ripple carry adder in a TMR configuration is used as the reference design. The Kogge-Stone adder, which is classified as a parallel prefix adder, has a critical path on the order of $\log_2 N$ (where N is the width of the adder in bits). The regularity of its structure makes it suitable for VLSI designs as well as FPGA implementations. This research is performed in two parts. First is to evaluate the previous work of K. Roy's group on fault tolerant adders meant for VLSI design [5] and by implementing this approach on FPGAs. Second is to investigate and design fully fault tolerant Kogge-Stone adders on FPGAs. The fault tolerant adders which are coded in VHDL are synthesized and implemented on the FPGAs. The functionality of the designed fault tolerant adders are studied and then compared with the base reference TMR adder in terms of delay and usage of resources as a function of their bit-widths. An optimal fault tolerant design adds little overhead to the system. The method of study for this research involves designing carry tree adders of varying widths up to 256 bits. The designs are synthesized by coding with VHDL using Xilinx's ISE 12.4 software. The performance metrics of timing delay and operational cost are observed from the synthesis reports. The functionality of the designed adders are

verified and simulating with ISIM. The critical delays of the designed adders are measured using a high-speed logic analyzer.

1.5 Thesis Outline

An outline of the thesis is as follows. Chapter Two describes the background work on fault tolerance in FPGAs. Chapter Three discusses the results of simulations of the basic fault tolerance adders using ISIM. Chapter Four introduces advanced fault tolerant concepts and then analyzes the performance metrics like timing (speed-adder delay) and the functionality based on the results obtained from the logic analyzer. Finally Chapter Five provides the conclusions and describes potential future work in this area.

Chapter Two

Fault Tolerance on FPGAs

2.1 Introduction

An adder plays a vital role in many digital circuit designs including Digital Signal Processors (DSPs) and microprocessors. The fault tolerant techniques for high speed adder designs are considered in this chapter. Triple Modular Redundancy (TMR), which is a common fault tolerance approach, is used as the base reference design. Advanced concepts for fault tolerant adders on FPGAs include roving, and graceful degradation. This chapter describes the design of the ripple carry adder, parallel prefix adders and then the fault tolerant concepts that can be applied to these designs.

2.2 Basic Adder Designs

This section describes the basic adders used in this thesis. The adders implemented on FPGAs are the ripple carry adder, Kogge-Stone adder, and sparse Kogge-Stone adders. The ripple carry adder is one of the simplest adder designs. The Kogge-Stone adder is an example of a parallel prefix adder. The internal blocks used in the adder designs are described in detail in this section.

2.2.1 Full Adder

A full adder is a circuit which adds three one bit binary numbers and outputs two one bit binary numbers. The block diagram is shown in Figure 2.1. Here, a and b are the two adder inputs and c_{in} is the carry input. The two outputs produced are the sum s and carry c_{out} . Table 2.1 depicts the truth table of the full adder. The following are the Boolean expressions for the full adder.

$$s = a \oplus b \oplus c_{in} \quad (2.1)$$

$$c_{out} = a \cdot b + b \cdot c_{in} + a \cdot c_{in} = a \cdot b + (a \oplus b) \cdot c_{in}$$

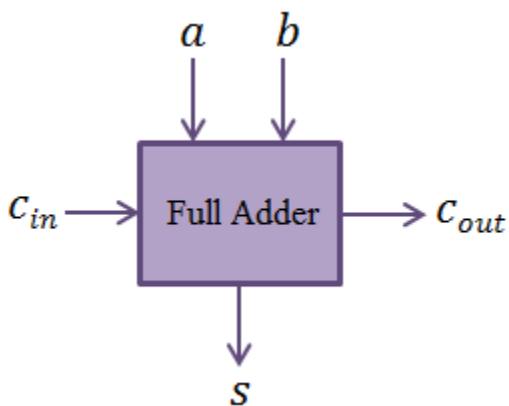


Figure 2.1: Block diagram of a full adder

For prefix adders, it is convenient to define the intermediate signals generate, propagate, and delete given by g , p , and d , respectively,

$$g = a \cdot b \quad (2.2)$$

$$p = a \oplus b$$

$$d = \overline{ab}$$

The sum and carry out are then given by,

$$s = p \oplus c_{in} \quad (2.3)$$

$$c_{out} = g + p \cdot c_{in}$$

Table 2.1. Truth table of a full adder

a	b	c_{in}	s	c_{out}	Carry Status
0	0	0	0	0	delete
0	0	1	1	0	delete
0	1	0	1	0	propagate
0	1	1	0	1	propagate
1	0	0	1	0	propagate
1	0	1	0	1	propagate
1	1	0	0	1	generate
1	1	1	1	1	generate

2.3 Ripple Carry Adder

The ripple carry adder is one of the simplest adders. It consists of a cascaded series of full adders. For example, a 4-bit adder can be constructed by cascading four full adders together as shown in Figure 2.2. The ripple carry adder is relatively slow as each full adder must wait for the carry bit to be calculated from the previous full adder.

The worst case delay of a ripple carry adder occurs when c_{in} propagates from the first stage to the most significant bit position. The delay for an N-bit adder is given by,

$$t_{adder} = (N - 1)t_{carry} + t_{sum} \quad (2.4)$$

where, t_{carry} is the carry propagation delay for one stage and t_{sum} is the time required to compute the sum bit for one stage. Hence, the delay of the ripple carry adder is of order N .

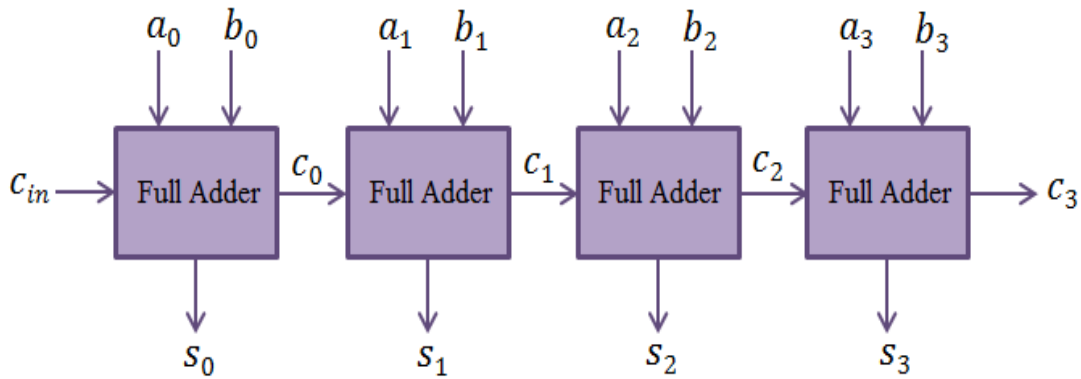


Figure 2.2: 4-bit ripple carry adder

2.4 Kogge-Stone Adder

The Kogge-Stone adder is classified as a parallel prefix adder since the generate and the propagate signals are precomputed. In a tree-based adder, carries are generated in tree and fast computation is obtained at the expense of increased area and power. The main advantage of this design is that the carry tree reduces the logic depth of the adder by essentially generating the carries in parallel. The parallel-prefix adder becomes more favorable in terms of speed due to the $O(\log_2 n)$ delay through the carry path compared to $O(n)$ for the RCA. The Kogge-Stone adder is widely used in high-performance 32-bit,

64-bit, and 128-bit adders as it reduces the critical path to a great extent compared to the ripple carry adder.

The operation of the tree-based adder can be understood using the concept of the fundamental carry operation (*fco*). This operator works on the generate and propagate pairs as defined by,

$$(g_L, p_L) \circ (g_R, p_R) = (g_L + p_L \cdot g_R, p_L \cdot p_R) \quad (2.5)$$

where g_L, p_L are the left input generate and propagate pairs and g_R, p_R are the right input generate and propagate pairs to the cell. For example, in a 4-bit carry lookahead adder, the carry combination equation can be expressed as,

$$c_4 = (g_4, p_4) \circ [(g_3, p_3) \circ [(g_2, p_2) \circ (g_1, p_1)]] \quad (2.6)$$

$$= (g_4, p_4) \circ [(g_3, p_3) \circ [(g_2 + p_2 \cdot g_1, p_2 \cdot p_1)]]$$

:

:

$$= g_4 + p_4 \cdot g_3 + p_4 \cdot p_3 \cdot g_2 + p_4 \cdot p_3 \cdot p_2 \cdot g_1$$

Since the *fco* obeys the associativity property, the expression can be reordered to yield parallel computations in a tree based structure [7],

$$c_4 = [(g_4, p_4) \circ (g_3, p_3)] \circ [(g_2, p_2) \circ (g_1, p_1)] \quad (2.7)$$

2.4.1 8-bit Kogge-Stone Adder

The 8-bit Kogge stone adder will be explained in detail in this subsection. An 8-bit Kogge-Stone adder is built from eight generate and propagate (GP) blocks, eight black cells (BC) blocks, eight gray cell (GC) blocks, and nine sum blocks as shown in the Figure 2.3. The details of the various blocks used in the structure of Kogge-Stone adder are discussed below.

1) GP block

The generate and propagate block takes a pair of operand bits (a, b) as inputs and computes a pair of generate and propagate signals (g, p) as output, as depicted in Figure 2.3. The output from the GP block is given by the equation (2.2).

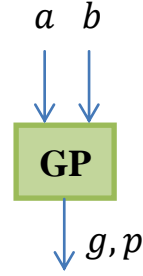


Figure 2.3: Generate-Propagate block

2) BC block

The black cell takes two pairs of generate and propagate signals (g_i, p_i) and (g_j, p_j) as input and computes a pair of generate and propagate signals (g, p) as output. It is shown in the Figure 2.4(a).

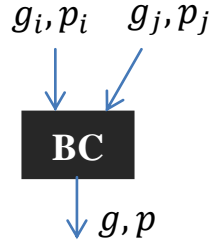


Figure 2.4 (a): Black cell

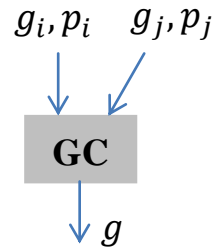


Figure 2.4 (b): Gray cell

The expressions for the output signals g, p generated by the black cell are given by

$$\begin{aligned} g &= g_i + p_i \cdot g_j \\ p &= p_i \cdot p_j \end{aligned} \tag{2.8}$$

3) GC block:

The gray cell takes two pairs of generate and propagate signals (g_i, p_i) and (g_j, p_j) as inputs and computes a generate signal g as output which is shown in Figure 2.4(b).

The expressions for the output signal g obtained by the gray cell is given a

$$g = g_i + p_i \cdot g_j \quad (2.9)$$

4) Buffer

The buffer takes a pair of the generate and propagate signals (p_i, g_i) as input and passes the same signals to the output. It is shown in Figure 2.5.

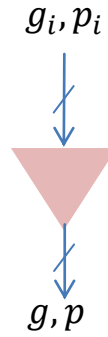


Figure 2.5: Buffer

The expressions for the output signals g, p obtained by the buffer block are given as

$$g = g_i \quad (2.10)$$

$$p = p_i$$

The complete schematic for the 8-bit Kogge-Stone adder is shown in Figure 2.6. An 8-bit Kogge-Stone adder is built from eight generate and propagate (GP) blocks, twelve black cell (BC) blocks, eight gray cell (GC) blocks, and eight sum blocks. To be aesthetic, an extra column has been added in our design to show the computation of c_8 . In practice, c_8 is generated by just adding an extra gray cell in the last column.



Figure 2.6: 8-bit Kogge-Stone adder

Higher Order Kogge-Stone Adders

Table 2.6. summarizes the various types of cells required for the Kogge-Stone adders with larger bit widths.

Table 2.6. Kogge-Stone adders of different bit widths

Bit Width of Kogge-Stone Adder	No. of GP Blocks	No. of Black Cells	No. of Gray Cells	No. of Sum blocks
16-bit	16	37	16	16
64-bit	64	257	64	64
128-bit	128	641	128	128
256-bit	256	1537	256	256

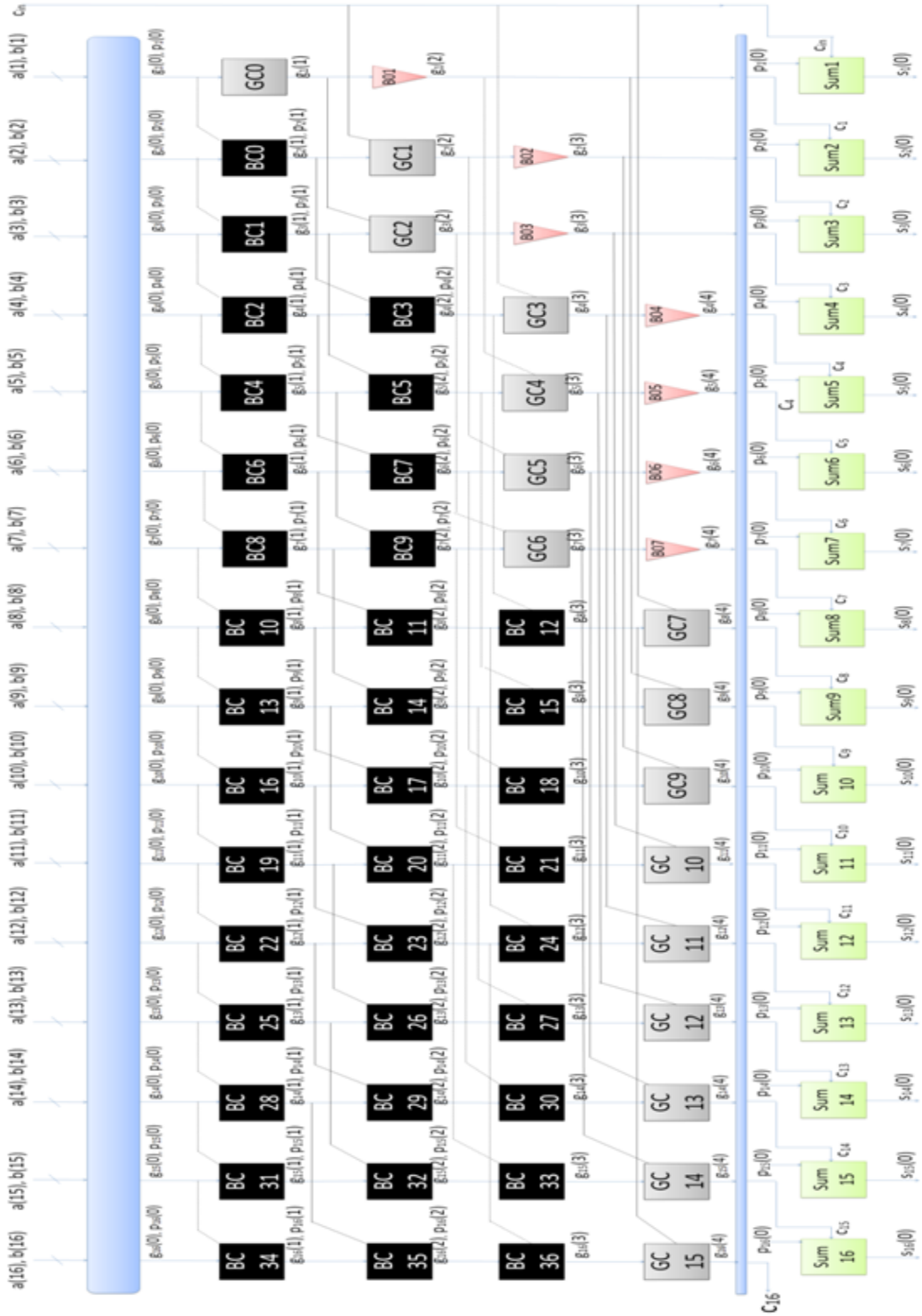


Figure 2.7: 16-bit Kogge-Stone adder [8]

2.5 Sparse Kogge-Stone Adder

The sparse Kogge-Stone adder consists of several smaller ripple carry adders (RCAs) on its lower half and a carry tree on its upper half. Thus, the sparse Kogge-Stone adder terminates with RCAs. The number of carries generated is less in a sparse Kogge-Stone adder compared to the regular Kogge-Stone adder. The functionality of the GP block, black cell and the gray cell remains exactly the same as in the regular Kogge-Stone adder. The schematic for a 16-bit sparse Kogge-Stone adder is shown in Figure 2.8. Sparse and regular Kogge-Stone adders have essentially the same delay when implemented on an FPGA although the former utilizes much less resources [9].

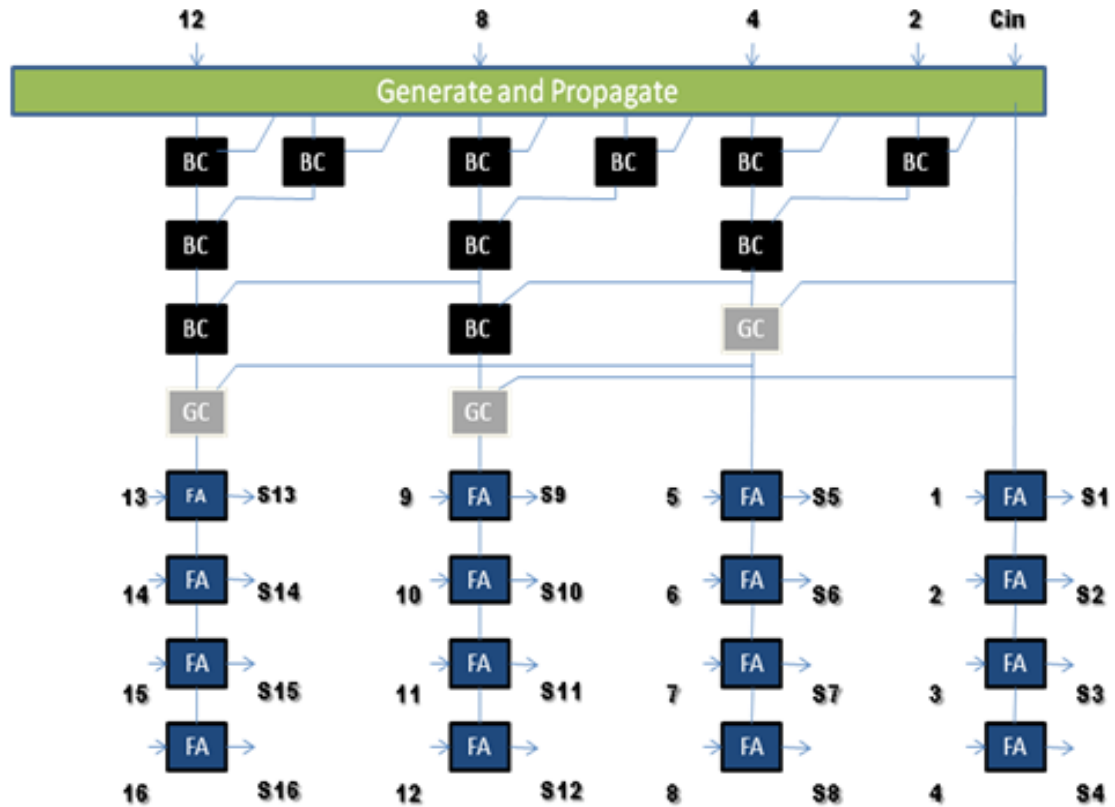


Figure 2.8: Sparse Kogge-Stone adder.

2.6. Basic Fault Tolerance - Hardware Redundancy

Basic fault tolerance can be achieved by N-module redundancy (NMR) where N refers to the degree of redundancy used in the design. This approach is easy to apply but results in high area overhead. For example, Triple Modular Redundancy (TMR) is a fault tolerant method where the hardware is essentially replicated in triplicate with a voter circuit used to pass the majority rule signals to the output. TMR is one of the most common methods used to create fault tolerant designs in both ASIC and FPGA implementations.

The general TMR is shown in Figure 2.9. Three copies of the same circuit are connected to a majority voter which is used to obtain the fault free output. This method works as long as all the faults are confined to one of the redundant blocks. The latency will be increased because of the voter in the circuit's critical path. The triple modular redundant ripple carry adder (TMR-RCA) is used as the reference design for this thesis. This adder is the simplest approach for both detecting and correcting faults. The block diagram of the TMR adder circuit using the ripple carry adders is shown in Figure 2.10.

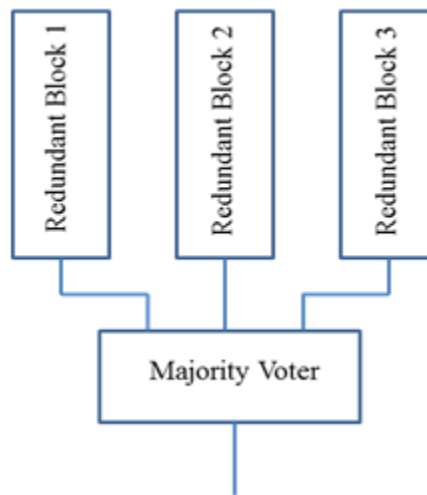


Figure 2.9: General Triple Modular Redundancy

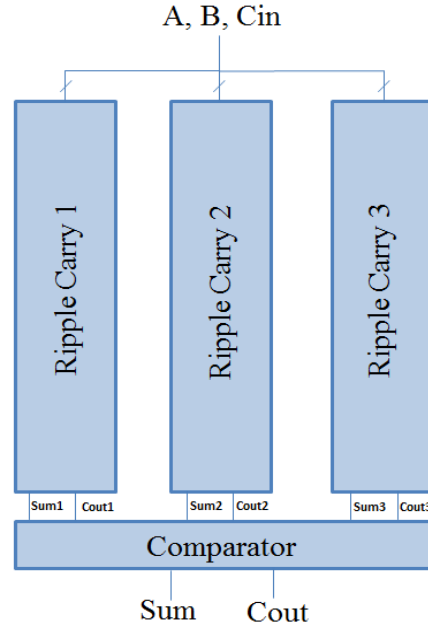


Figure 2.10: TMR adder circuit using ripple carry

2.7 Advanced Fault Tolerant Methods

Compared to the basic TMR-RCA, more advanced fault tolerant methods exist including roving and graceful degradation approaches. Allowing fault tolerance to operate at different levels of abstraction might facilitate a more cost-effective design [10]. Fault tolerance can be classified into three categories, namely information redundancy, time redundancy, and a hybrid approach. The technique involved in information redundancy includes the use of error-correcting codes. Time redundancy tradeoffs area for time of the available time slot and recomputes in a different time slot resulting in low overhead but longer delay [11]. Hybrid approaches make use of several types of the available tree [5] for redundancy to achieve fault tolerance. This section focuses on a hybrid approach to achieve fault tolerance. For example, a fault tolerant design can be implemented by taking advantage of the inherent redundancy of a parallel prefix adder like the Kogge-Stone adder which is described next.

2.7.1 Structural Design – Hybrid Approach

A fault tolerant parallel prefix adder can be implemented using a Kogge-Stone adder due to the inherent redundancy in the carry-tree [5]. The even and the odd carry trees present in the adder are mutually exclusive as shown in the Figure 2.11. An extra

column is added to make sure that if any fault occurs at the last sum bit then it can be restored. This approach falls under the hybrid category due to its inherent structural redundancy and use of time redundancy since two clock cycles are required to correct a

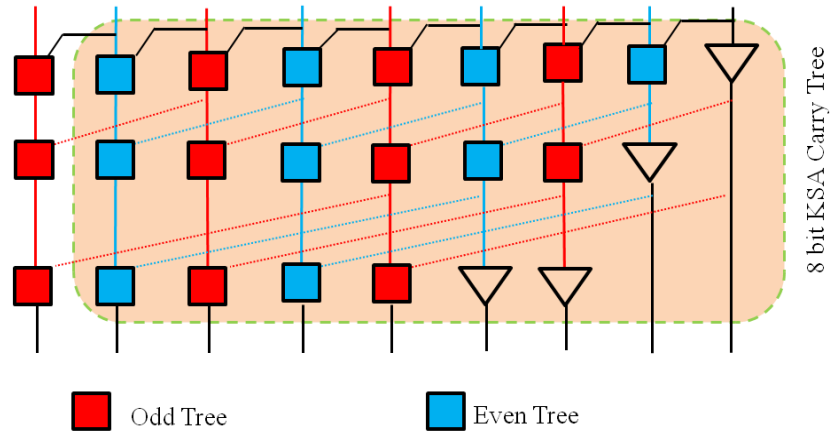


Figure 2.11: An 8 bit Kogge-Stone carry tree illustrating the mutually exclusive even and odd carry trees.

fault. Due to the mutual exclusive nature, if a defect is present in the one-half of the carry tree, the other half can be utilized to compute the carries for both the even and odd carries. The timing diagram for this design is illustrated in Figure 2.12. In the scenario depicted, three instructions are scheduled on three different adders present in the execution unit. The second adder is defective and is evaluated in two clock cycles whereas the fault free adders are evaluated in a single-clock cycle, assuming that the defect is in the odd bits. In the timing diagram adaptive clocking is performed during the execution of the second instruction for correct functionality of the pipeline. In cycle-3 the odd bits are computed correctly and stored as the operands are left shifted by one bit. The even bits are discarded in cycle-3. As the second adder which is scheduled for instruction 2 is faulty, it will be completely evaluated only at the end of cycle-3 even though it is initiated at cycle-2. The even bits are computed in cycle-2 and registered while the odd bits are discarded.

Thus the output will be produced only after two clock cycles. In cycle-1, one of the correct set of bits which are either even or odd are computed and stored at the output

registers. The operands are shifted by one bit and the remaining sets of bits are computed and stored in cycle-2. The trade-off in time may be acceptable in some circumstances using proper scheduling and micro-architectural changes, as detailed in [5]. The design for the error correcting 8-bit Kogge-Stone adder is shown in Figure 2.13. The multiplexers at the inputs are used for shifting the operands left by one bit whereas the multiplexers at the output are used for shifting the partially correct sums to the right by one bit.

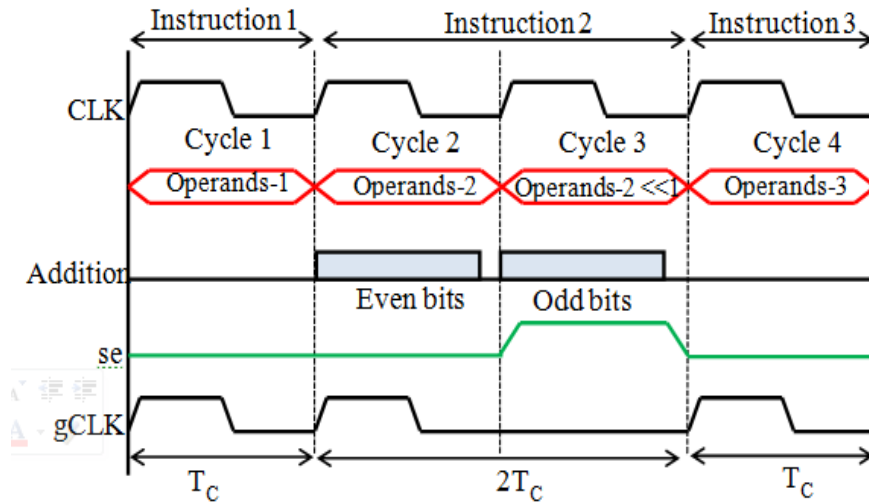


Figure 2.12: Timing diagram for three adders in execution unit (T_C is the clock period)

2.7.2 Roving

Roving is one of the fault detection methods detailed in [4]. An important work in this area is described by Emmert, Stroud and Abramovici in which the testing and the diagnostic process takes place in designated Self-Testing Areas (STARs) of an FPGA, without disturbing the normal operation of the system [12]. Roving performs a progressive scan of the FPGA structure by swapping blocks of equivalent functionality for testing. In the roving detection of faults, the FPGA is split into equal-sized regions in which one region is configured to perform self-test, while the remaining areas carry out the designed function of the FPGA. An important advantage is that the detected fault does not affect the working logic of the system [6]. As a result, the operation does not have to be interrupted for fault diagnosis. It relies on incremental run-time configuration, which is the ability to dynamically reconfigure part of an FPGA without actually disturbing the operation performed in the rest of the device.

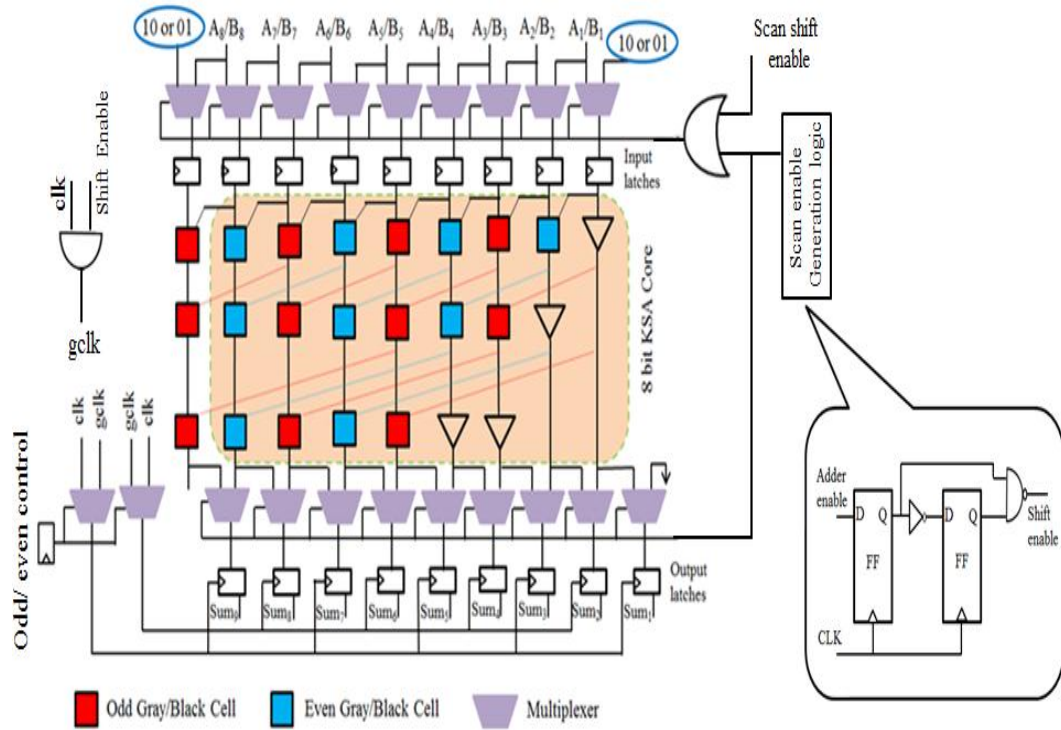


Figure 2.13: Block diagram for the proposed 8-bit fault tolerant Kogge-Stone adder [5].

The overhead will be less in roving compared to other redundancy methods as the overhead consists of one-self test region and a controller which manages the reconfiguration process. A disadvantage is that roving results in longer signal delays and may force a reduction in the system clock speed as the connections of the adjacent functional areas are stretched [12]. Roving spares and fault scanning is illustrated in Figure 2.14.



Figure 2.14: Roving area under test across the chip.

2.7.3 Graceful Degradation

This method involves the use of redundant hardware for both the detection and recovery from faults. Graceful degradation occurs when one of the spare hardware blocks is used to replace a faulty one, resulting in a degradation of system functionality.

During fault detection, each block is checked with the voter circuit against two identical blocks used as spares. If there is any error then the faulty block will be replaced with one of the spare blocks. Figure 2.15 shows the general description of a system that exhibits graceful degradation. In this example, there are four operational blocks with two test blocks.

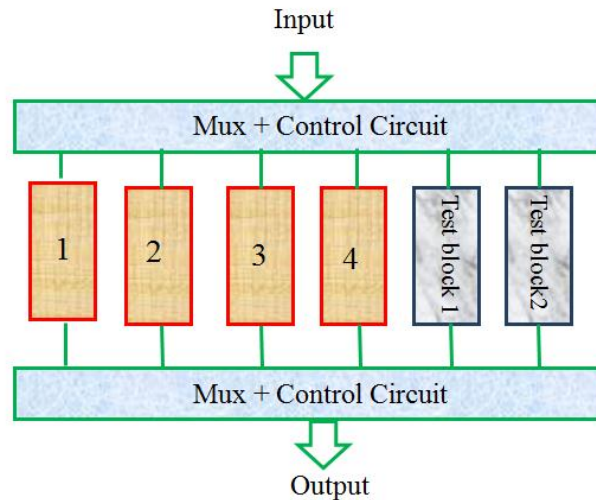


Figure 2.15: General block diagram of graceful degradation

In this scheme, blocks 1 to 4 and the two test blocks have identical functionality. Two redundant blocks, labeled Test block 1 and 2 are used to check the blocks labeled 1 to 4 in sequence. A TMR-like checking scheme is utilized in this example. If an error is found, then the faulty block is replaced with one of the test blocks. Thus, the system remains operational but the fault checking capability is degraded.

Figure 2.16 provides a detailed example. Consider the case where the fault is present in block 4. Figure 2.16(a) shows the first block being checked with the test blocks. As no error is found in block 1, the fault checking now proceeds to block 2, shown in Figure 2.16(b). There is no error in block 2. Block 3 is then compared with the test blocks shown in the Figure 2.16(c). Finally checking goes to the block 4 as no fault is

present in block 3 (see Figure 2.16(d)). As block 4 is faulty, the entire block is replaced with one of the test blocks, shown in Figure 2.16(e). The system remains functional, but the ability to detect existing faults has been degraded, since a TMR-like fault checking scheme can no longer be utilized.

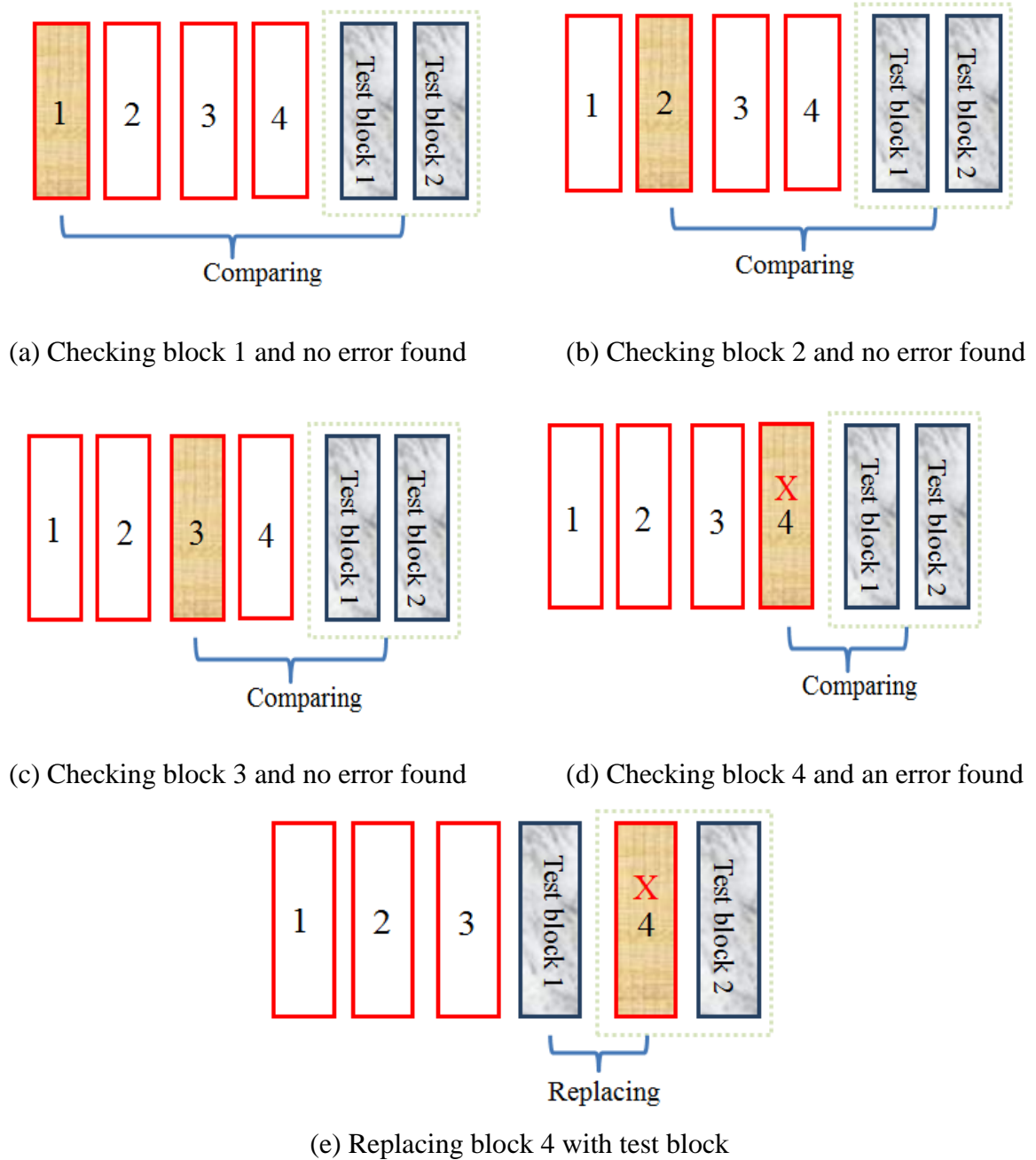


Figure 2.16: General view of the graceful degradation process.

Summary

In this chapter, the general fault tolerance methods implemented on electronic systems like Triple Modular Redundancy (TMR) were discussed. Also advanced fault tolerant methods like the hybrid approach with structural redundancy, the roving concept and the graceful degradation approach were explained. In the next chapter, the methods of implementation on an FPGA, including TMR-RCA, and the error correcting structural approach [5] are discussed. A proposed fault tolerant sparse Kogge-Stone adder is also introduced.

Chapter 3

Basic Fault Tolerant Implementation

3.1 Introduction

Some basic methods for implementing fault tolerant adder designs are described in this chapter. First the FPGA implementation method using Xilinx's Integrated Software Environment (ISE) software is described. The simulation results for the Triple Modular Redundancy-Ripple Carry adder (TMR-RCA) and the error correcting regular Kogge-Stone adder are detailed next. Then the design used for creating a partially fault tolerant lower half sparse Kogge-Stone adder which includes the use of ripple carry adders is described.

3.2 FPGA Implementation Method

Xilinx's ISE Design Suite 12.4 software is used to implement all the fault tolerant adder designs on the Spartan 3E FPGA. The design flow is outlined in Figure 3.1. A project navigator helps to manage the entire design process which includes design entry, simulation, synthesis and implementation by downloading the configuration onto the FPGA device.

First, the design flow begins by creating a new project in Xilinx's ISE, followed by coding the model in VHDL. The VHDL code for the select fault tolerant adders is given in the Appendices. The code is then synthesized using the ISE software. The functionality of the designs are verified by creating a test bench in VHDL and then simulating with the ISIM tool. Finally, the functionality of the implemented adder is verified using a TLA 7012 Logic Analyzer.

3.3 Triple Modular Redundancy-RCA

The simulation results for the TMR-RCA structure discussed in Chapter Two are described in this section. Input stimuli were carefully selected to demonstrate the functionality of each adder. Specific cases are discussed for each adder approach. The VHDL code for the 32-bit TMR-RCA and higher bit widths implemented on hardware

are given in Appendix A. A signal named **fault** is used to inject a fault into one of the RCAs and the signal **error** goes high when a fault is detected.

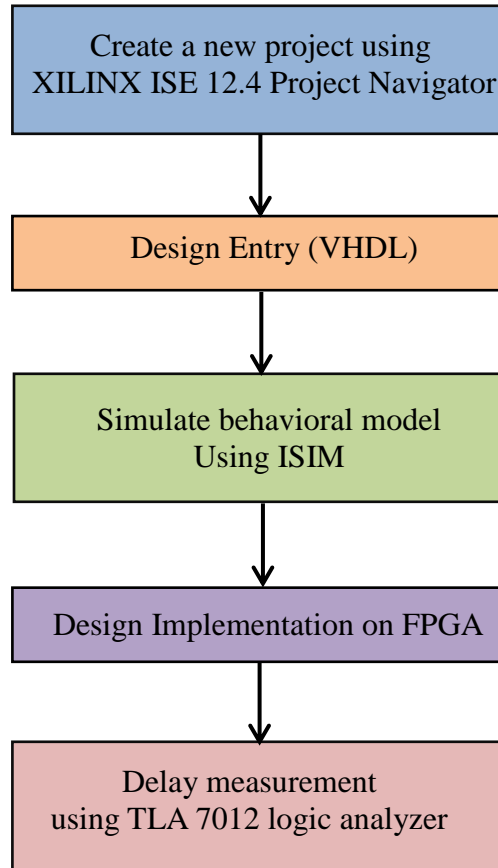


Figure 3.1: General design flow

The simulation results for the adders of widths up to 256 bits are observed in this thesis. The simulation results for the 64-bit TMR-RCA are shown in the Figure 3.2 as an example. The simulation results are shown in two parts: (a) the actual functionality of the adder and (b) the fault tolerant operation.

(a) Consider the cycle with index = 3. The inputs are $c_{in} = 1$, $a = '0000a0000000000000'$, and $b = '000000000000000000'$. There is no fault for any of the ripple carry adders in this case and the outputs signals $sum1$, $sum2$ and $sum3$ from all three ripple carry adders are the same. As expected the output from the comparator is $s = '0000a0000000000001'$.

(b) Consider the case where a fault is injected into one of the ripple carry adders at index = 5. The inputs are $c_{in} = 1$, $a = '0000000070000000'$ and $b = '0000000100000000'$. The resulting output sum s equals $'0000000170000001'$ which is chosen from the majority of the redundant RCA blocks, $sum1$ or $sum2$, as $sum3$ is incorrect. Hence the ability of this adder to recover from the embedded fault is demonstrated.

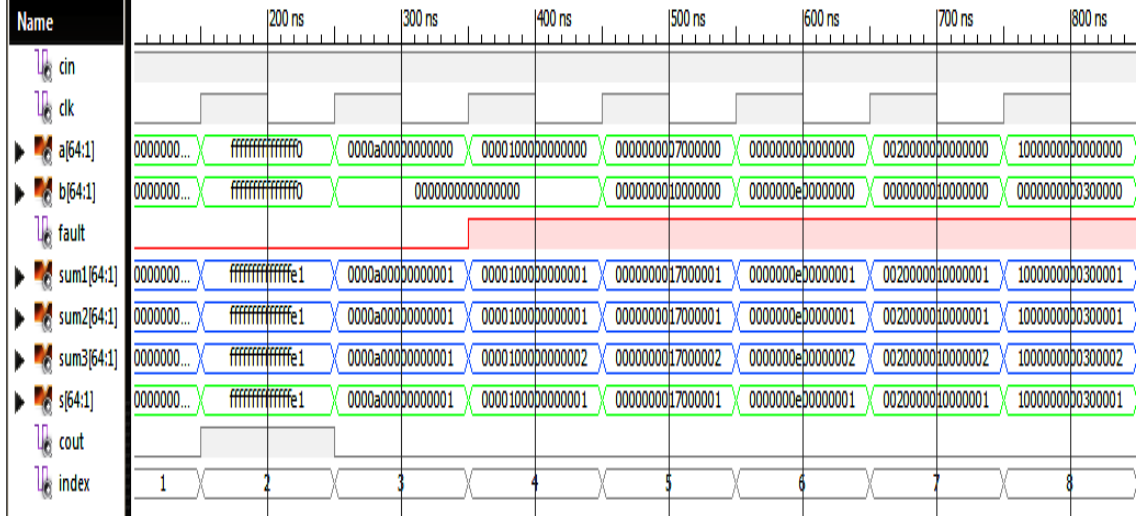


Figure 3.2: Simulation results for the 64-bit TMR-RCA

3.4 Regular Kogge-Stone Adder Fault Correction Approach

The simulation results for the regular Kogge-Stone adder error correcting approach discussed in chapter 2 is described in this section. For the 64-bit Kogge-Stone adder, different values are assigned to the inputs a , b and c_{in} for each clock cycle by synthesizing a test bench. The VHDL code for error correction on a Kogge-Stone adder with a width of 64 bits is given in Appendix B.

The simulation results of the error correcting 64-bit regular Kogge-Stone adder are shown in the Figure 3.3. The simulation results are depicted in two parts: (a) the actual functionality of the adder and (b) fault tolerant operation.

(a) Consider cycle 1 in Figure 3.3. The inputs are $a = 'ffffffffffff'$, $b = '0000000000000000'$. After two clock cycles the resulting sum $s[64:0] = '1ffffffffffffe'$ is obtained. The correct sum is taken by ignoring the last bit i.e. $s[64:1]$ as the operands are shifted by one bit and the remaining sets of bits are computed and stored in cycle-2.

(b) Consider cycle 3. The inputs assigned are $a = \text{'ffffffffffffffff'}$, $b = \text{'ffffffffffffffff'}$ and the signal named **fault** is used to inject a fault into the adder. As expected, the output sum should be $s[64:0] = \text{'1fffffffffffffff'}$ which is obtained after two clock cycles. The correct sum is taken by ignoring the last bit i.e. $s[64:1]$. Hence the ability of this adder to recover from the embedded fault is demonstrated.

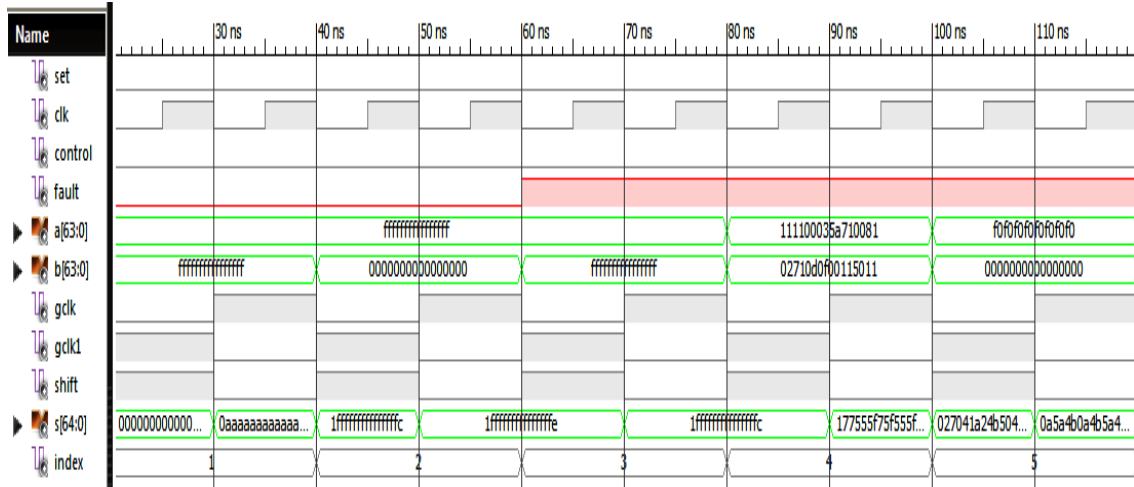


Figure 3.3: Simulation results for the 64-bit error correcting Kogge-Stone adder.

3.5 Lower Half Fault Tolerant Sparse Kogge-Stone Adder

This section describes the design of a Sparse Kogge-Stone adder which is both fault detecting and fault correcting. In the sparse Kogge-Stone adder, ripple carry adders are required at the output, whose length is dependent on the degree of the sparseness in the carry tree. Figure 2.9 in chapter two illustrates a sparse Kogge-Stone adder with a factor of four reduction in the carry tree. Thus, it needs four ripple carry adders in its lower half.

Fully fault tolerant designs for the sparse Kogge-Stone adder have been achieved in two steps. First a fault tolerant design for the lower half (i.e., the ripple carry adder chains) is implemented. Then the design is extended to make the upper half (i.e, the carry tree) fault tolerant. This is discussed in Chapter Four.

As the design contains the addition of the ripple carry adders in the sparse Kogge-Stone adder, a similar testing methodology can be used as with the TMR-RC adder to detect and correct errors found in any of the ripple carry adders. An illustration of the design for the fault tolerant sparse Kogge-Stone adder is shown in Figure 3.4.

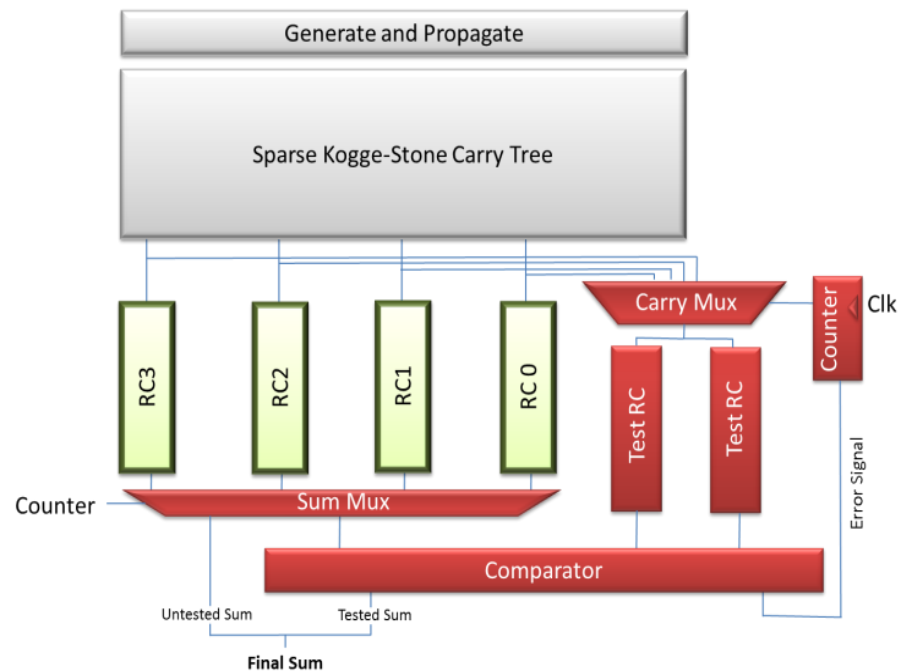


Figure 3.4: Block diagram of fault tolerant sparse Kogge-Stone adder

The highlighted portion in red in Figure 3.4 is the error correction and detecting logic. Two extra ripple carry adders must be added to the design for testing (Test RC) which is similar to the TMR-RCA. Also, some multiplexers and a bit counter are required. During each clock cycle, one of the four ripple carry adders is selected for testing. The corresponding carry-in and A and B operands are routed through the multiplexer denoted as Carry Mux to the TestRCs. The selections of these inputs are controlled by a counter which is driven by the clock. The final evaluation is performed by a comparator in which the outputs of the tested RCA branch (one of RC0 to RC3) are switched simultaneously. The final sum is obtained at the falling edge as the valid output has been passed through the tested sum.

The timing diagram for this design is illustrated in Figure 3.5. Once an error is detected, the clock operating the bit counter is stopped at the RCA branch where the error was detected and will continue correcting the faulty RCA branch until the error has been removed.

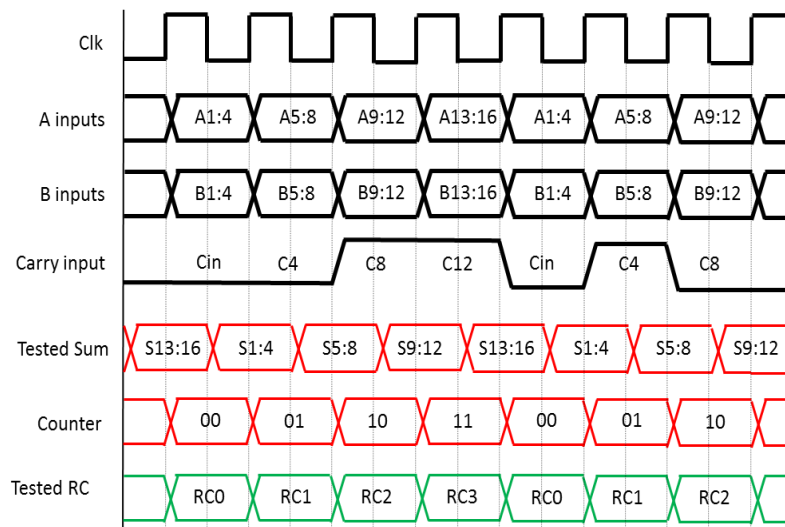


Figure 3.5: Timing diagram for the lower half fault tolerant Kogge-Stone adder

3.5.1 Simulations of the Sparse Kogge-Stone Adder

The design is coded in VHDL and simulated using ISIM. The following simulation results illustrate the successful detection and the correction of the fault in the adder. A signal named **fault** is used to inject a fault into one of the RCAs and the signal **error** goes high when a fault is detected. The synthesis results for the lower half fault tolerant sparse Kogge-Stone approach are obtained for the Spartan 3E FPGA. The VHDL code for a 32-bit lower half sparse Kogge-Stone adder is given in Appendix C.

The simulation results for the 64-bit sparse Kogge-Stone adder are shown in Figure 3.6. The simulation results are depict in two parts: (a) the functionality of the adder and (b) the fault tolerant operation.

- (a) Consider the case where the index = 1. The corresponding assigned inputs are $a = \text{'ffffffffffffffff'}$, and $b = \text{'ffffffffffffffff'}$. The final sum obtained at that clock cycle is correctly given as $sum = \text{'fffffffffffffffe'}$.
- (b) Consider the case where the fault is introduced at index =2. The corresponding assigned inputs are $a = \text{'0000000000000000'}$ and $b = \text{'ffffffffffffffff'}$. As described

earlier, the error is detected and the correct sum is obtained at the falling edge of $\text{index} = 2$. Thus the final sum obtained at the given inputs is $\text{sum} = \text{'ffffffffffffffff'}$. Hence the ability of this adder to recover from the embedded fault is demonstrated.

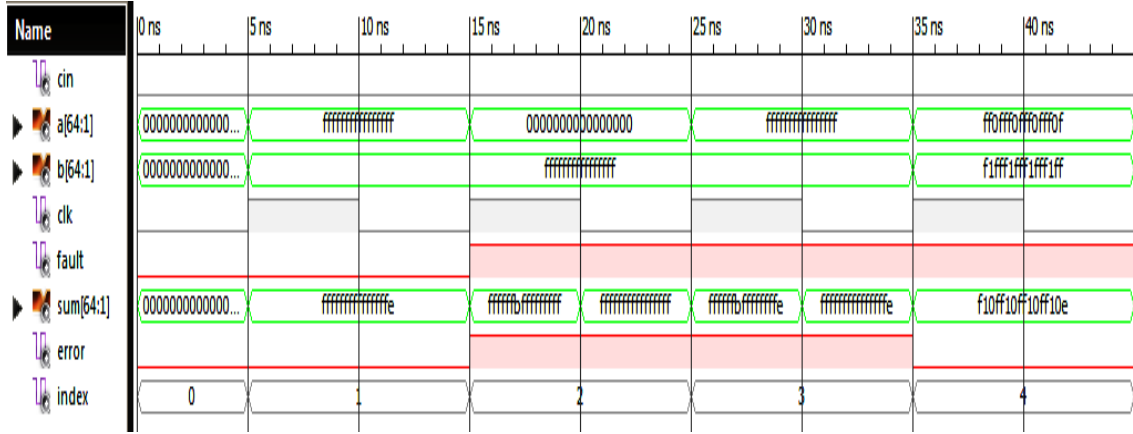


Figure 3.6: Simulation results for the 64-bit lower half FT sparse Kogge-Stone adder.

Summary

In this chapter, the simulation results for some basic fault tolerance techniques suitable for implementation on FPGAs were discussed. The inherent redundancy in the carry tree of the Kogge-Stone adder is used for error correction. Also by introducing additional ripple carry adders in the lower half of the sparse Kogge-Stone adder, fault tolerance can be achieved. In the next chapter, more advanced concepts for implementing fault tolerant adders on FPGAs are described. In addition, performance metrics like timing (speed-adder delay) and resource utilization for the different methods are compared. The functionality of the various fault tolerant adder designs is verified using a logic analyzer for testing these implementations.

Chapter 4

Advanced Fault Tolerance Concepts

4.1 Introduction

Having discussed the simulation results for basic fault tolerant adders, this chapter will describe the architectures for implementing advanced fault tolerant sparse Kogge-Stone adders. First, a fault tolerant carry tree which is designated as the upper half of the sparse Kogge-Stone adder is designed. Second, the permanent replacement of a faulty block with a spare block is implemented for the ripple carry adders present on the lower half of the Sparse Kogge-Stone adder. As this results in reduced fault checking ability, this process is known as graceful degradation. The design of these proposed fault tolerant approaches with their simulation results are detailed in this chapter.

4.2 Upper Half Fault Tolerant Sparse Kogge-Stone Adder

The fault tolerant design for the lower half of the sparse Kogge-Stone adder which consists of the ripple carry adder chains, was explained in Chapter Three. This section focuses on making the carry tree of the sparse Kogge-Stone adder, designated as the upper half, fault tolerant. In this design, the carry tree of the sparse Kogge-Stone adder is split into three sections, which are depicted by the following colors: (1) green, (2) purple and (3) blue in Figure 4.1. The testing methodology developed makes use of redundant carries generated by the carry tree (C_i-C) and the ripple carry adders (C_i-R). For example, for a 16-bit sparse Kogge-Stone adder with 4-bit RCAs, there are two sets of carries generated for C_4 , C_8 , and C_{12} (i.e., $i = 4, 8, 12$). The complete schematic for the upper half error detection scheme for the 16-bit sparse Kogge-Stone adder is shown in Figure 4.1.

The technique for detecting a fault is now explained in detail. For example, if carry C_4-C does not match carry C_4-R , the error must be located in the first section assuming the first ripple carry adder (RC_0) is fault free. If the result obtained is fault free, a fault free carry C_4 will enter the second ripple carry adder (RC_1).

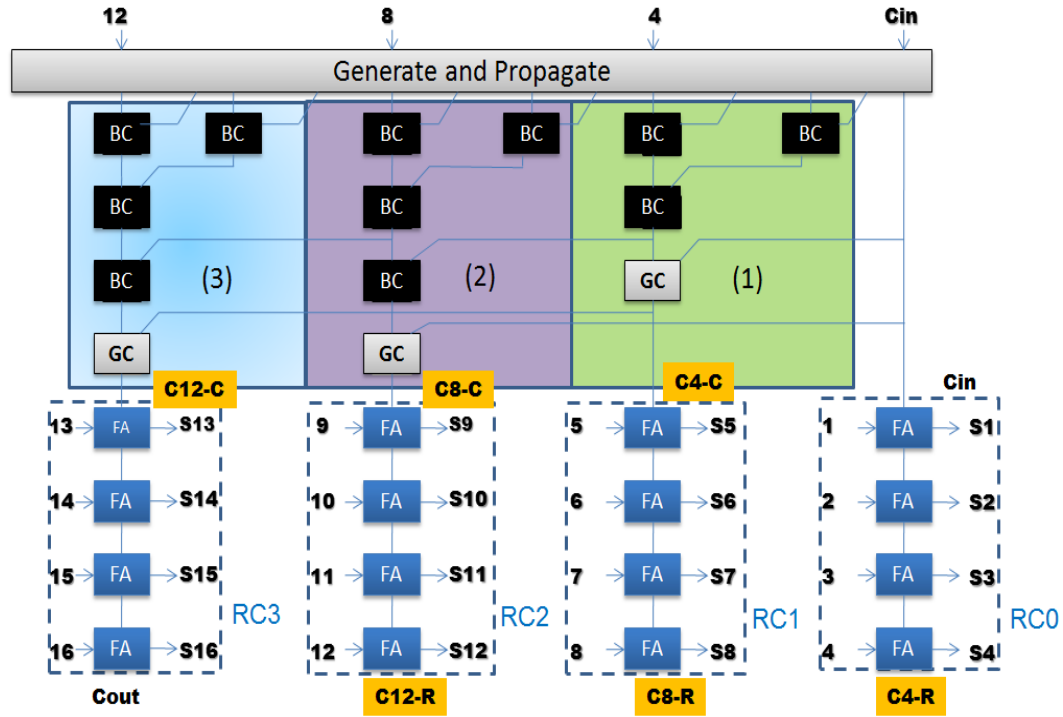


Figure 4.1: Upper half error detection scheme for 16-bit sparse Kogge-Stone

Next, if a mismatch is found in the C8-R and C8-C pair, there exists a fault in the second section. Finally, if the carries C8 and C4 are fault free, the error can be in the third section if a mismatch is found in the C12 pair. The fault detection mechanism is summarized in Figure 4.2. A mismatch in a carry pair is indicated by ‘1’ in Figure 4.2. If there are multiple faults at the same time this approach is still able to correct at its corresponding clock cycle.

X = don't care
1 = Carry Mismatch
0 = Carry Match

C12	C8	C4	Fault selection
X	X	1	Green (1)
X	1	0	Purple (2)
1	0	0	Blue (3)
0	0	0	No Error

Figure 4.2: Upper half detection truth table

Spares for each section can be made available to replace a faulty section by using multiplexers to reroute the carry tree from the faulty branches to the spare section. A

method for increasing the number of sections detectable is possible with the adoption of a time redundant ripple carry adder which is already in the bottom-half test circuit. By feeding the carry out produced by the first ripple carry (C4-RCO) into the test ripple carry adders (Test RCs), the adder produces completely fault free values for comparison over the course of three clock cycles. These fault free carries can then be compared to the carry tree's output for fault detection in the five sections as illustrated in the Figure 4.3. An example depicted in Figure 4.3 shows the fault originating in the red section (4) and passing through C12 and C4. As C8 shows no signs of a fault in this example, the only possible case for this error combination can be traced back to a fault in section (4). Similarly, with this scheme a list of possible combinations was developed into a truth table as shown in Figure 4.4. The figure also includes an example of a time redundancy ripple carry circuit capable of producing the fault free carries necessary for the comparison. It takes three clock cycles to produce C12 as shown in Figure 4.4. If an error has been successfully isolated to a specific section of the adder, the faulty section can be replaced using a decoder and multiplexer, which reroutes the signals to the replacement sections built into the design.

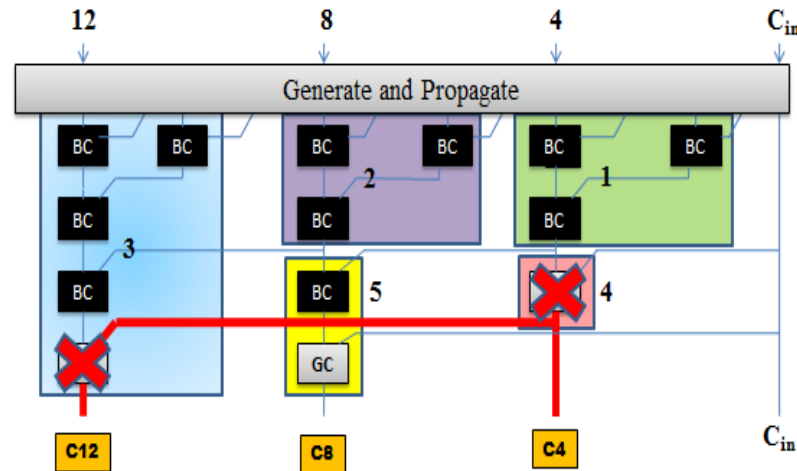


Figure 4.3: Upper half detection scheme with fault free carry comparisons.

The fault can only be corrected after a faulty sum value has been allowed to pass. Once the rerouting of the replacement section has completed, the adder will perform normally as seen in the simulation results in the next section.

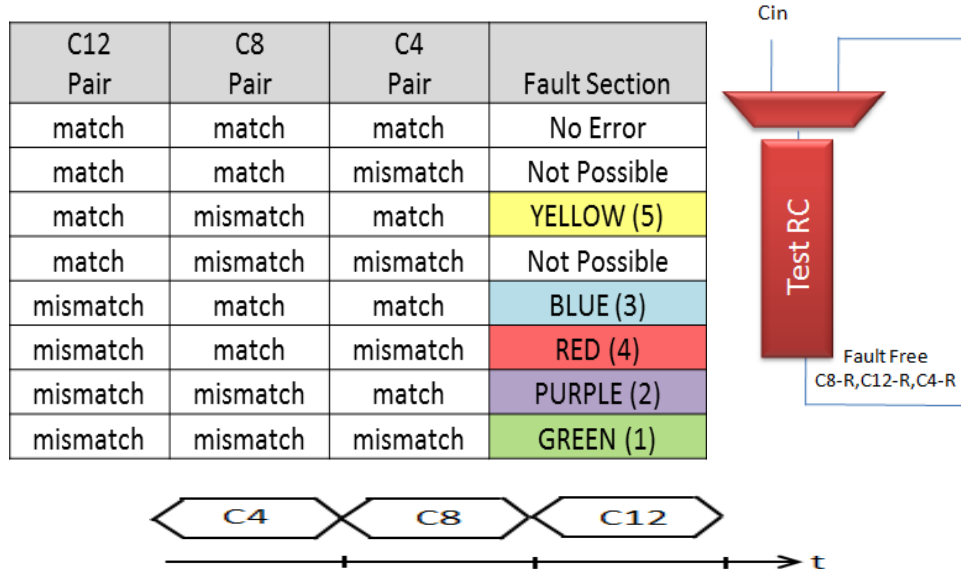


Figure 4.4: Truth table for upper half error detection with fault free comparisons

A similar fault detection scheme is applied for the sparse Kogge-Stone adder of higher bit widths. The complete schematic for the proposed upper half error detection scheme for a 32-bit sparse Kogge-Stone adder is shown in Figure 4.5. For example, for a 32-bit sparse Kogge-Stone adder with 4-bit RCAs, there are two sets of carries generated for C8, C16, and C24. The same technique for detecting a fault as in the 16-bit sparse Kogge-Stone is now explained for this 32-bit adder. For example, if carry C8-C does not match carry C8-R, the error must be located in the first section assuming the first ripple carry adder (RC0) is fault free. If the result obtained is fault free, a fault free carry C8 will enter the second ripple carry adder (RC1). Next, if a mismatch is found in the C16-R and C16-C pair, there exists a fault in the second section. Finally, if the carries C16 and C8 are fault free, the error can be in the third section if a mismatch is found in the C24 pair.

This illustrates the proposed scheme for implementing fault tolerance in the carry tree and can easily be extended to higher bit width sparse Kogge-Stone adders.

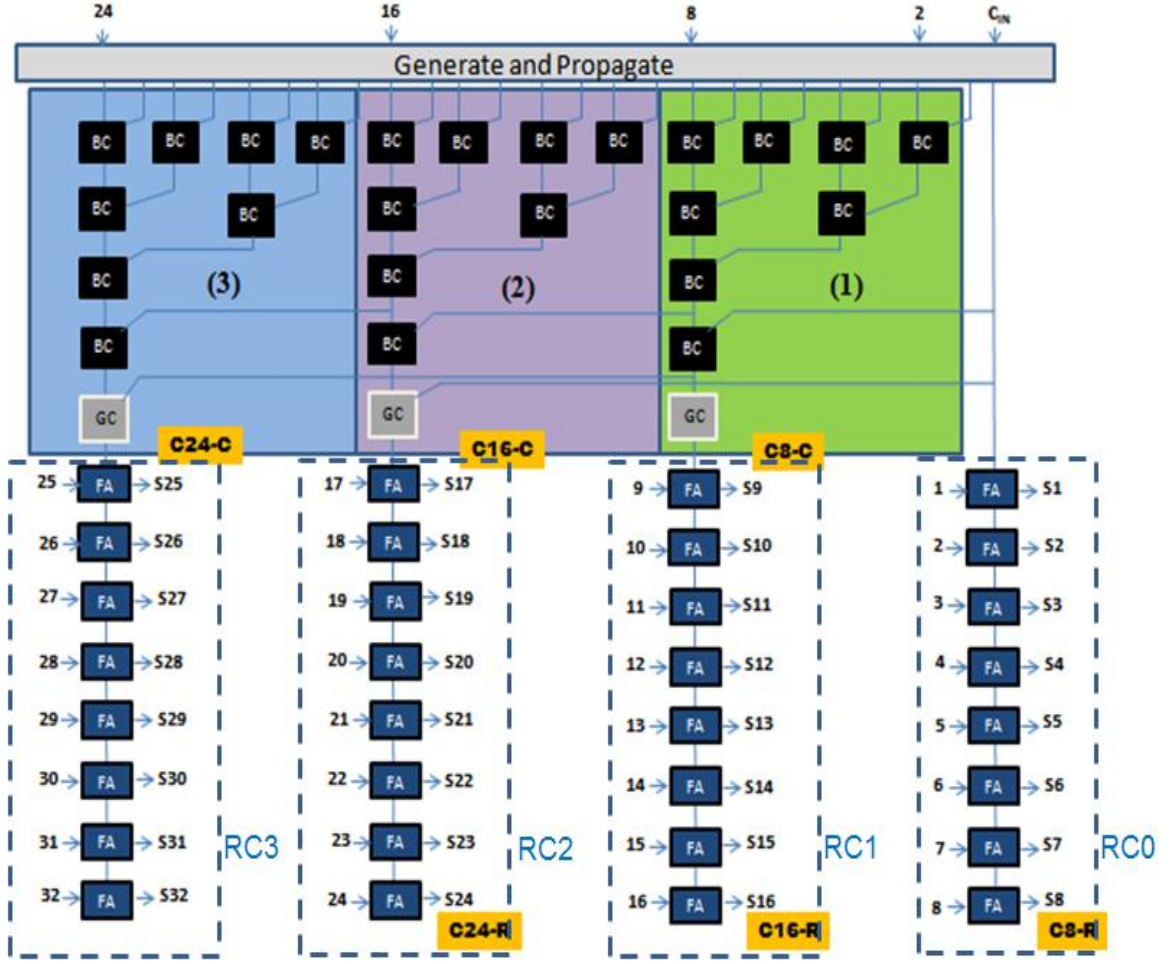


Figure 4.5: Upper half error detection scheme for a 32-bit sparse Kogge-Stone adder

4.2.1 Simulation Results

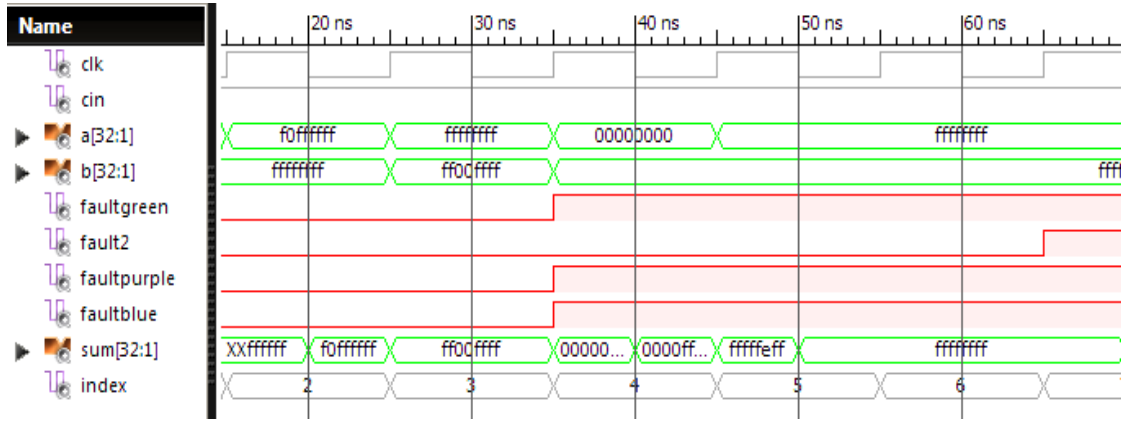
The final design is coded in VHDL and simulated using ISIM. The VHDL code for a 32-bit sparse Kogge-Stone upper half approach is given in Appendix D. A signal named **faultgreen** is used to inject a fault into the section (1) and a signal named **fault2** is injected into the backup of section (3) colored in blue in Figure 4.5. The synthesis results for the 32-bit fault tolerant sparse Kogge-Stone upper half approach are obtained for the Spartan 3E FPGA and its simulation results are shown in Figure 4.6.

The simulation results are shown in two parts: (a) functionality of the adder and (b) fault tolerant performance.

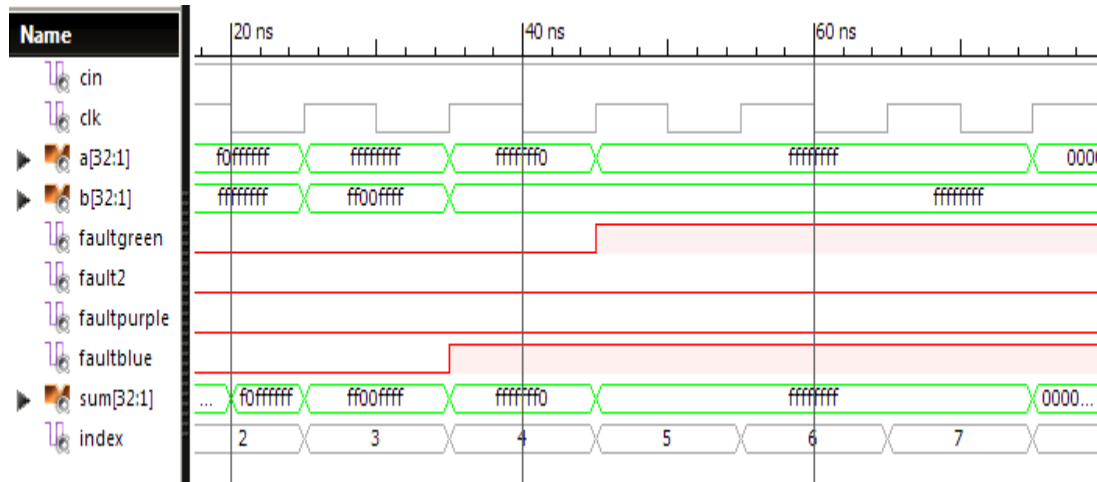
(a) Consider the cycle with index = 3 shown in Figure 4.6 (a). The inputs are $c_{in} = 1$, $a = \text{'ffffff'}$, and $b = \text{'ff00fff'}$. The correct output $sum = \text{'ff00fff'}$ is obtained.

- (b) Consider the cycle with index = 4 shown in Figure 4.6 (b). The chosen inputs are $c_{in} = 1$, $a = \text{'ffffff0'}$, and $b = \text{'ffffff}'$. The signal named **faultblue** is introduced in section (3) of the design. This tests the adders ability to detect and correct the fault. The obtained output sum is correctly computed as 'ffffff0' .
- (c) Consider the cycle with index = 5 shown in Figure 4.6 (b). The chosen inputs are $c_{in} = 1$, $a = \text{'ffffff}'$, and $b = \text{'ffffff}'$. In this case signals, named **faultblue** and **faultgreen** are used to inject a fault into sections (3) and (1), respectively. The obtained output sum is correctly computed as 'ffffff' .

The simulations results are also checked and verified for various other fault combinations (see Appendix G).



4.6 (a): Normal adder operation of the sparse Kogge-Stone upper half



4.6 (b): Fault tolerant adder operation of the sparse Kogge-Stone upper half

Figure 4.6: Simulation results for 32-bit sparse Kogge-Stone upper half approach

4.3 Graceful Degradation

Graceful degradation is a process of permanently replacing the faulty block with the test block, thus degrading the fault tolerant capability of the circuit in order to continue its primary function. The generic example with a complete block diagram was explained in Chapter Two.

4.3.1 Implementation

This section describes the proposed graceful degradation approach for the sparse Kogge-Stone adder. Multiplexers and a bit counter are added to this design. An illustration of this design can be seen in Figure 4.7, where the error correction and the detection logic is highlighted in green. In this design, RC0 to RC3 are the ripple carry adders present on the lower half of the sparse Kogge-Stone adder and an extra ripple carry adder called RCSpare, is added for replacement of a faulty RCA.

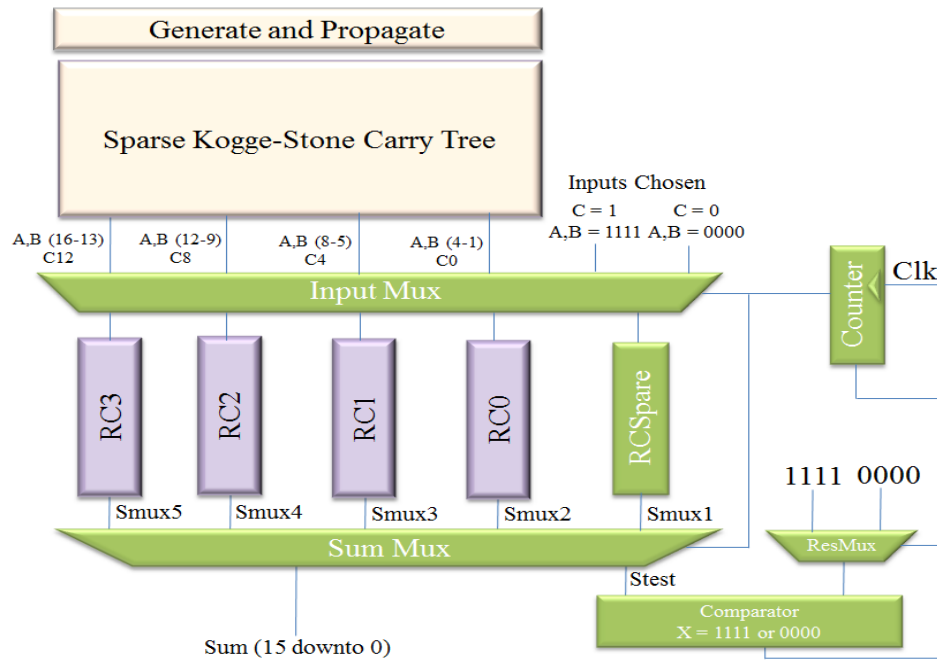


Figure 4.7: Block diagram of graceful degradation on the sparse Kogge-Stone adder

During each clock cycle, one of the four ripple carry adders, RC0 to RC3 (highlighted in purple in Fig. 4.7) is selected for testing. Two test input cases are chosen

for testing the selected adder, and while it is being tested its corresponding inputs are routed to RCSpare. The results obtained for the given inputs are expected earlier and fed to the multiplexer called ResMux. Thus ResMux selects one of the expected results for the test inputs. The output obtained from the ripple carry adder which is under test is then compared with the expected output from the ResMux using a comparator. If there is no error while comparing then the checking continues to the other inputs chosen, i.e. operands A, B = 0000 and C = 0. Thus the expected outputs for the two chosen cases are compared with the obtained output from the selected ripple carry adder under test. The test inputs are chosen for testing the ripple carry adder based on the popular *Stuck-At* fault model [13]. A stuck-at 0 fault is tested using inputs of 1111 and a stuck-at 1 fault is tested by setting the inputs to 0000. If the selected ripple carry adder passes the two test cases it is considered to be fault free.

The counter is driven by a clock and controls the selection of these inputs. The checking then continues to the next ripple carry adder. Once an error is detected, the clock operating the bit counter is stopped and the block is permanently replaced with the replacement block RCSpare while the adder can continue to function correctly. As further checking is not possible, fault tolerant capability is lost. However this can be remedied by adding more replacement blocks. This is a tradeoff between fault tolerant capability and area.

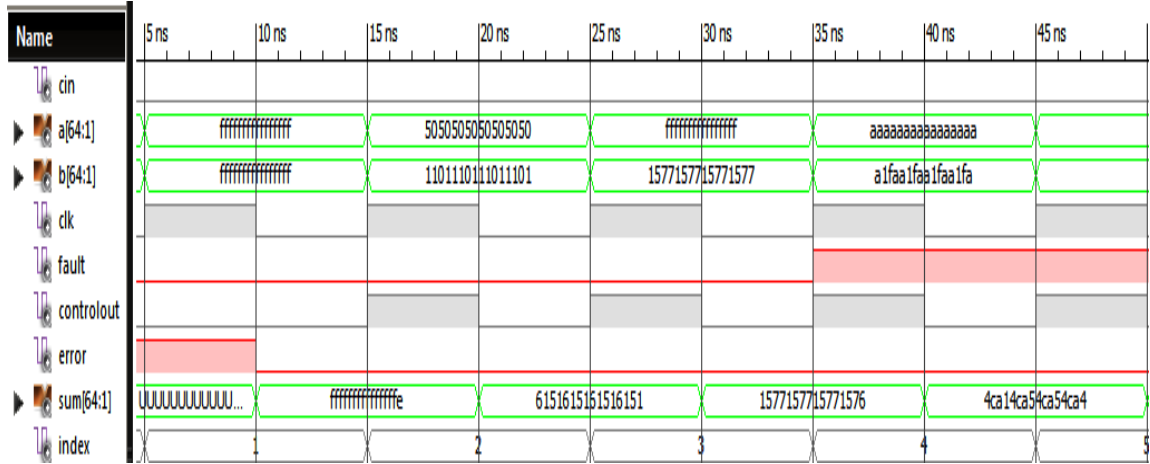
4.3.2 Simulation Results

The final designs are coded in VHDL and simulated using ISIM. The VHDL code for a 32-bit TMR-RCA and graceful degradation implemented on hardware are given in Appendix E. A signal named **fault** is used to inject a fault into one of the RCAs and the signal **error** goes high when a fault is detected. The synthesis results for fault tolerant sparse Kogge-Stone lower half approach are obtained for the Spartan 3E FPGA.

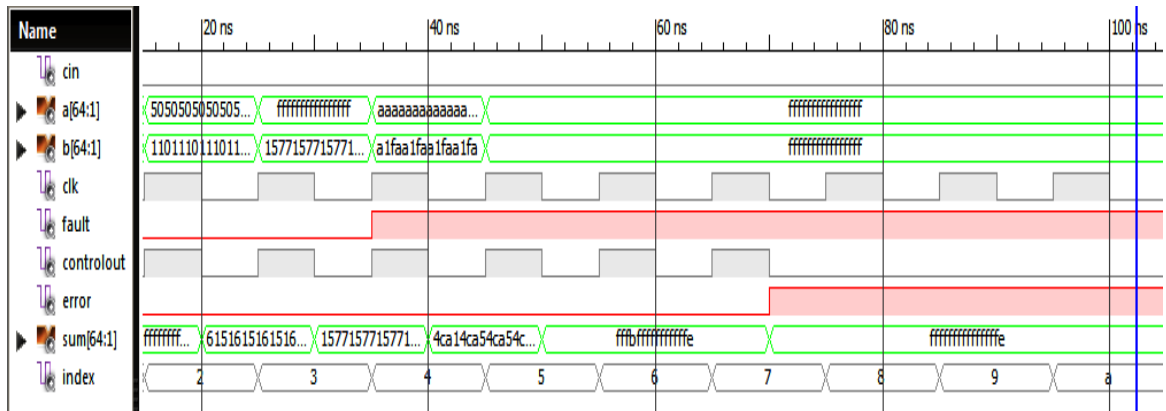
The simulation results of the 64-bit graceful degradation on a sparse Kogge-Stone adder are shown in the Figure 4.8. The simulation results are shown in two parts: (a) functionality of the adder and (b) the fault tolerant performance.

(a) Consider the cycle with index = 1 shown in Figure 4.8 (a). The inputs are $c_{in} = 0$, $a = \text{'ffffffffffffffff'}$, and $b = \text{'ffffffffffffffff'}$. The resulting output is $sum = \text{'ffffffffffffffe'}$.

(b) Consider the cycle with index = 7 shown in Figure 4.8 (b) for worst case fault correction. The chosen inputs are $c_{in} = 0$, $a = \text{'fffffffffffffff'}$, and $b = \text{'fffffffffffffff'}$. The signal named **fault** is used to introduce an error into the adder logic. As the resulting output $\text{sum} = \text{'fffffffffffffff'}$, the fault tolerant performance is demonstrated.



4.8 (a): Normal adder operation of the graceful degradation adder



4.8 (b): Fault tolerant adder operation of the graceful degradation adder

Figure 4.8: Simulation results for a 64-bit graceful degradation sparse Kogge-Stone adder

4.4 Synthesis Results

The synthesis results for a TMR-RCA, regular Kogge-Stone fault tolerant adder, proposed fault tolerant sparse Kogge-Stone for both lower half and upper half, and graceful degradation are obtained for a Spartan 3E FPGA. Design statistics are obtained by synthesizing the adders using Xilinx *ISE* software in two ways. First, the number of resources in terms of look up tables (LUTs) is observed. The results obtained are shown in Figure 4.9 which gives an estimation of the number of the look up tables (LUTs) used by each design. The sparse Kogge-Stone upper half approach was implemented for 16 and 32 bits, thus demonstrating it can be scaled to wider bit widths. However, partitioning the sections of the sparse Kogge-Stone adder to higher order bit widths is quite a challenging task.

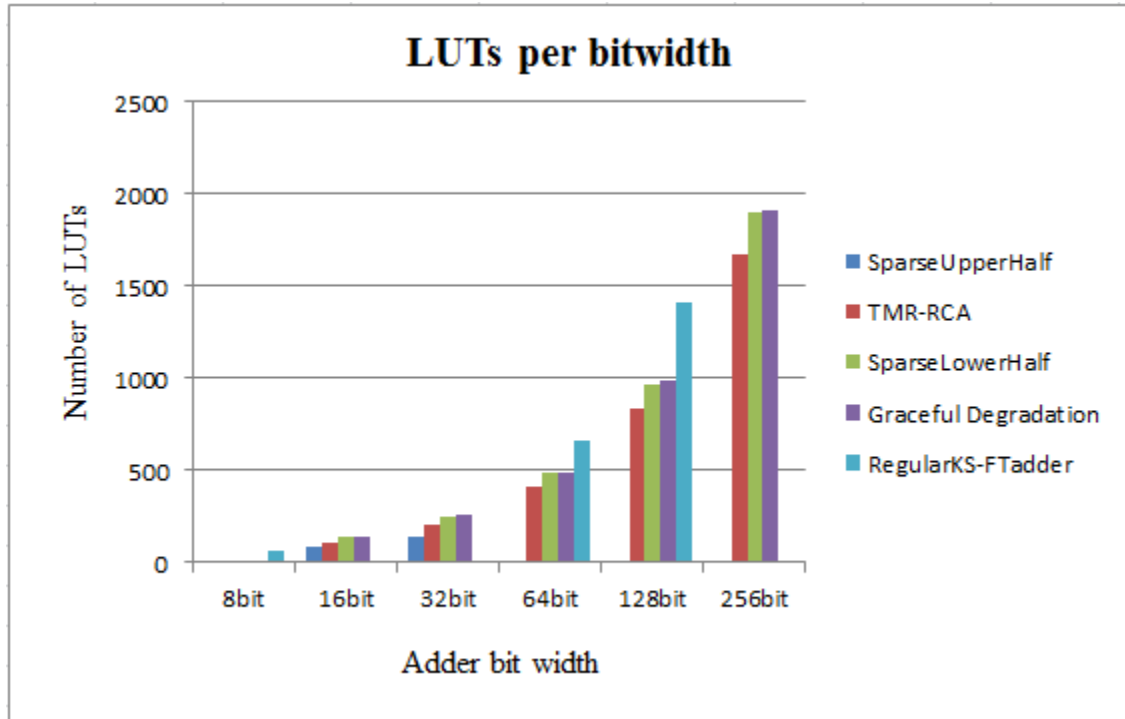


Figure 4.9: Estimation of resources used from FPGA synthesis

Second, the simulated delays for the fault tolerant adders are observed. The TMR adders seems to be the most efficient approach in terms of resources for fault tolerant design on FPGA due to its simplicity and the ability to take the advantage of the fast-carry chain.

The graphs are plotted for delay versus the bit width of the adders. The delay of each TMR-RCA is compared to the delay of the fault tolerant sparse Kogge-Stone adder for widths varying from 16 to 256 bits.

Figure 4.10 depicts the graph for the total adder delay taking each component into consideration. The comparator delay is constant for all bit widths. The carry tree delay has logarithmic characteristics whereas the ripple carry delay is linear as expected. Thus the overall total delay of the fault tolerant lower half sparse Kogge-Stone is the summation of all the individual component delays. Hence the obtained plot is logarithmic as expected for the sparse Kogge-Stone adder as shown in Figure 4.10.

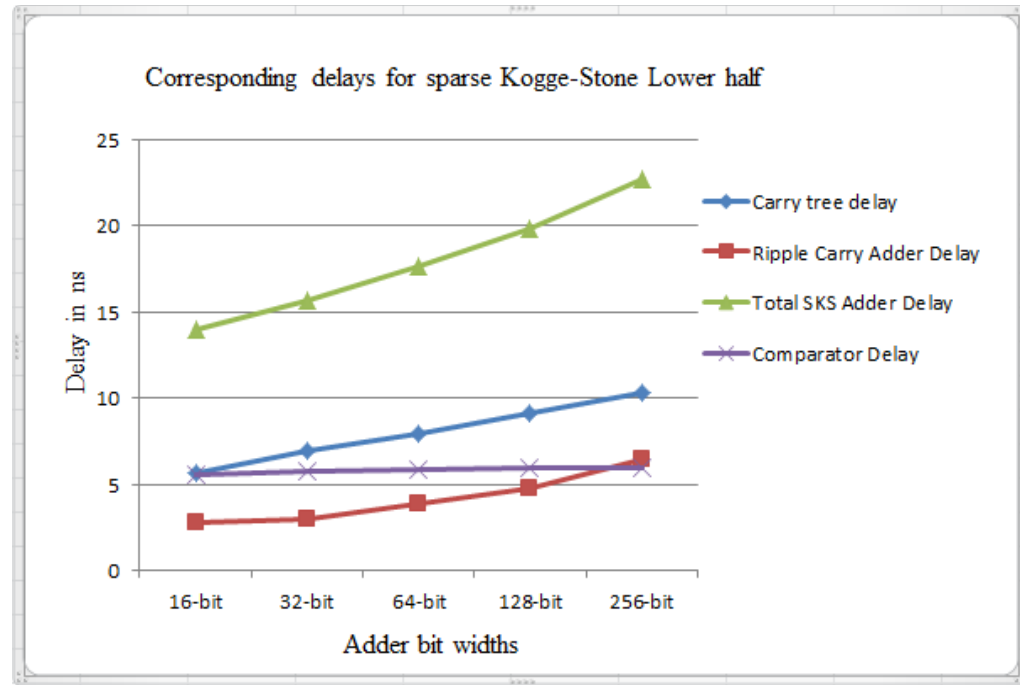


Figure 4.10: Corresponding delays for the sparse KS on Spartan 3E FPGA

Figure 4.11 depicts the graph for the delay of the adders on Spartan 3E FPGA. In the figure, the delay of the fault tolerant adders for different bit widths is shown. It can be observed from the graph that the delay is small for the TMR-RCA. Thus the TMR-RCA is still the best approach for an FPGA fault tolerant implementation at bitwidth of 128 and less due to its simple design of the approach and the use of the fast carry chain. At

widths of 256 bits, a graceful degradation approach that uses a sparse Kogge-Stone adder has smaller delay.

From the FPGA synthesis results shown in Figure 4.9, it is observed that the regular Kogge-Stone fault tolerant adder requires a lot more resources compared with the proposed fault tolerant approach on the sparse Kogge-Stone lower half and the graceful degradation adder. This tradeoff comes at the expense of a higher logic depth resulting in a longer critical path in the graceful degradation on the sparse Kogge-Stone adder than the regular Kogge-Stone fault tolerant adder. The tradeoffs for the proposed fault tolerant adders can be observed by looking at the delay plots. At 256 bits, the graceful degradation is better than the TMR-RCA due to its use of a sparse Kogge-Stone adder configuration.

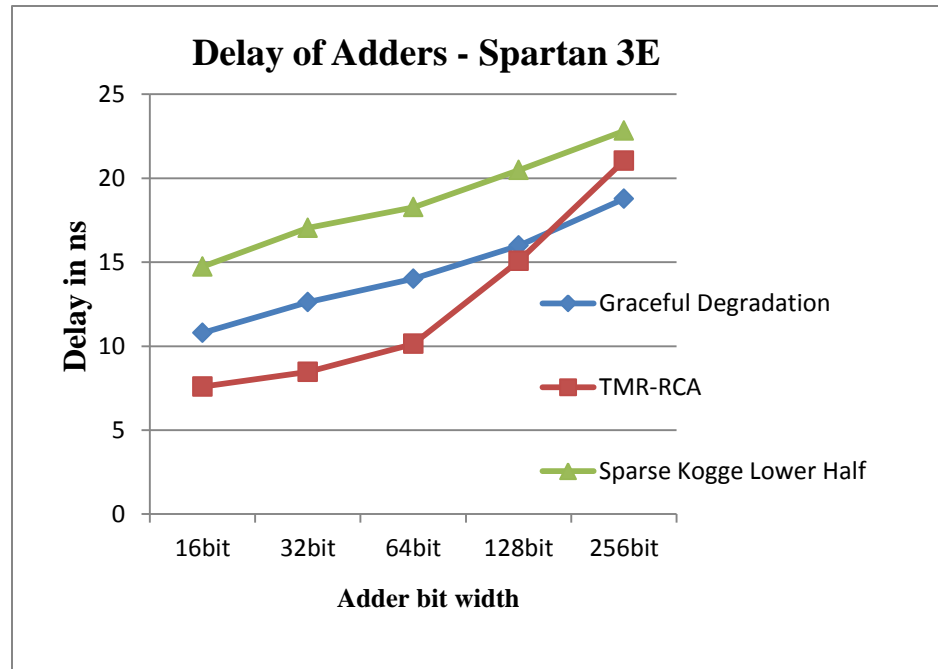


Figure 4.11: Delay of FT adders on Spartan 3E FPGA

The delays for the adders synthesized on Virtex-5 FPGA are shown in Figure 4.12. It can be observed that the overall delay of all the adders is roughly half that compared to the Spartan 3E FPGA. This is expected since the Virtex 5 FPGA is built using a more advanced process than the Spartan 3E FPGA. In addition, the each Virtex 5 logic cell uses 6-input look-up tables (LUTs) compared to the 4 input LUTs of the Spartan 3E

FPGA, resulting in a more efficient implementation overall. (See Appendix H for details).

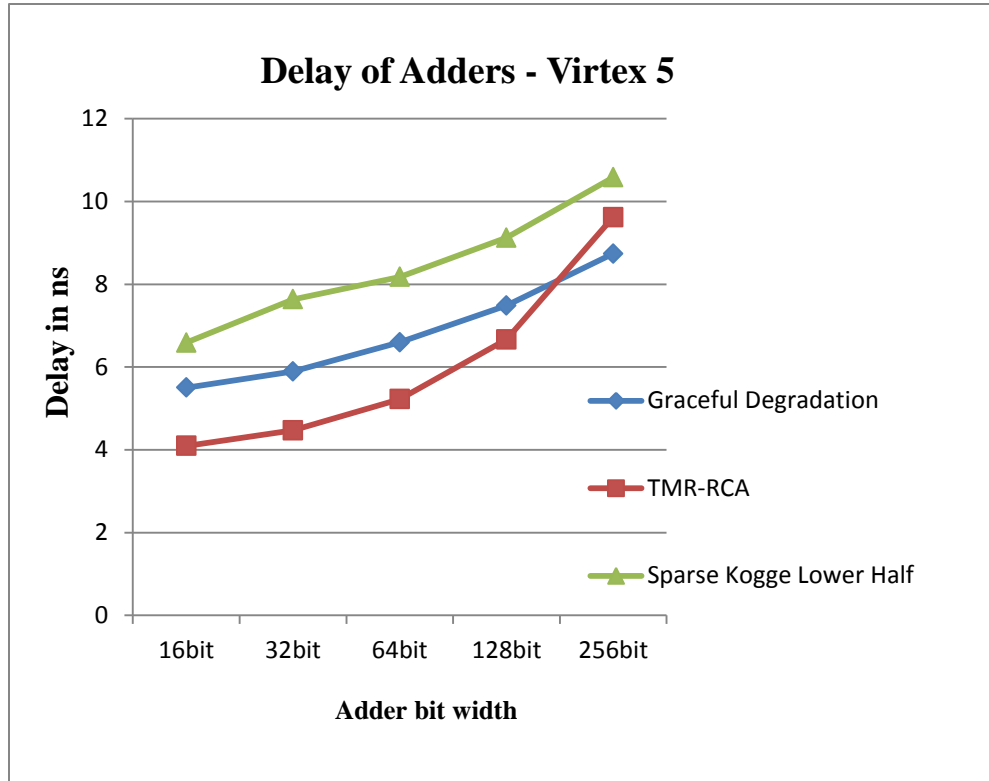


Figure 4.12: Delay of FT adders on the Virtex 5 FPGA

4.5 Hardware Implementation

After synthesizing all the designs using the Xilinx ISE software, the proposed fault tolerant adders TMR-RCA, sparse Kogge-Stone adder (lower half), and graceful degradation are then implemented on the Spartan 3E FPGA. The functional verification and the delays of all the structures are performed by using a high speed Tektronix Logic Analyzer (TLA). The VHDL code for a 32-bit adder implemented on hardware is given in Appendix F to keep it to a manageable size and all delay measurements here are shown for 64 bits. The resulting waveforms obtained by implementing a 64-bit TMR-RCA on Spartan 3E FPGA are shown in Figure 4.13.

For the TMR-RCA, different values are assigned for the inputs a , b and c_{in} for each clock cycle by synthesizing a test circuit along with the fault tolerant adder on the FPGA. These waveforms show how the critical adder delay is measured at every output

transition with respect to a clock signal on the logic analyzer. The corresponding delay obtained at particular transition is shown in Figure 4.13 which is highlighted in red (see Appendix I for the detailed procedure for measuring adder delay).

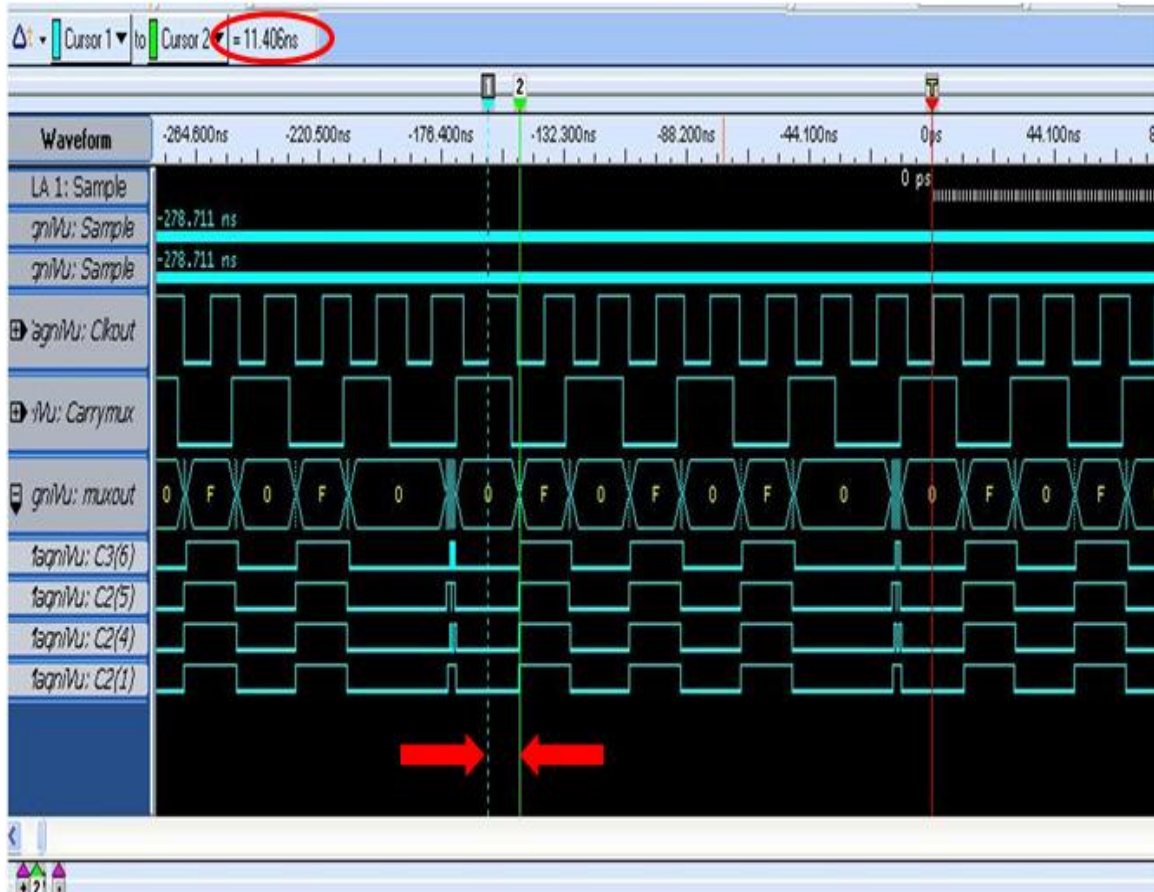


Figure 4.13: Measured delay for the 64-bit TMR-RCA

For obtaining the worst case delay for the fault tolerant lower half sparse Kogge-Stone adder, the fault has been introduced to one of the TestRC blocks as shown in Figure 4.14. This ensures that a comparison is made between one of the RCA blocks (RCA0 to RCA3) and the fault free TestRC block. The resulting waveforms obtained by implementing a 64-bit sparse Kogge-Stone lower half on the Spartan 3E FPGA are shown in Figure 4.15.

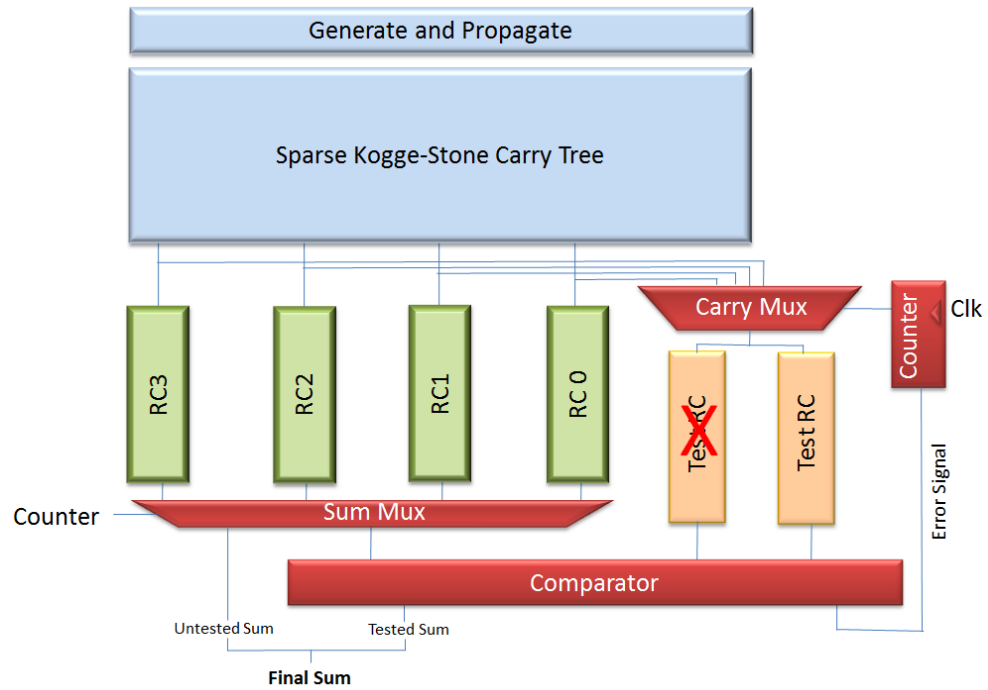


Figure 4.14: Implemented procedure for simulating the worst-case delay on the sparse Kogge-Stone lower half approach



Figure 4.15: Measured delay for the 64-bit sparse Kogge-Stone adder

The resulting waveforms obtained by implementing a 64-bit sparse graceful degradation on Spartan 3E FPGA are shown in Figure 4.16.

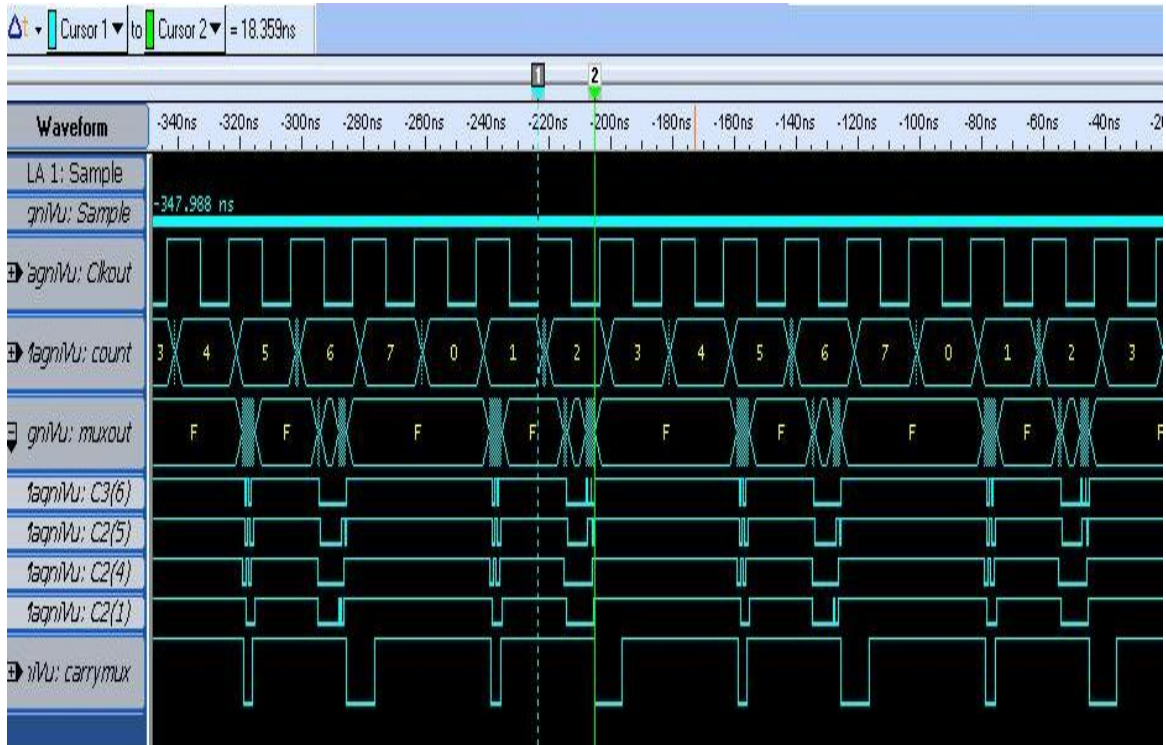


Figure 4.16: Measured delay for the 64-bit graceful degradation adder

The worst case delays are obtained from the logic analyzer for TMR-RCA, sparse Kogge-Stone adder (Lower Half), and graceful degradation for the corresponding chosen input patterns for the adders of different bit widths. Figure 4.17 summarizes the test results, showing that the measured results are faster but the overall relative delay between the adders is comparable to the results obtained from the synthesis reports.

Summary

This chapter has discussed the advanced techniques for implementing fault tolerant adders with its simulation results. By introducing backup sections in the upper half of the sparse Kogge-Stone adder, fault tolerance can be achieved. The key results are summarized in terms of utilization of resources and the corresponding adder delays.

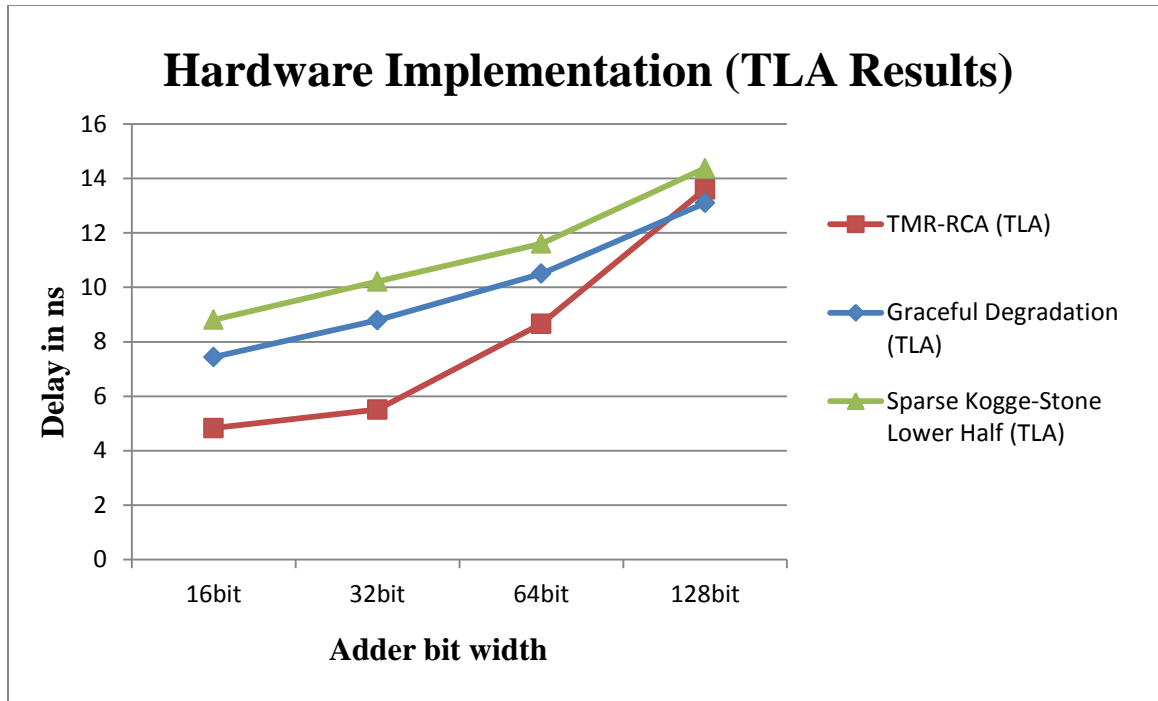


Figure 4.17: Summary of adder delays on Spartan 3E.

The functionality of the designs implemented on the hardware has been verified by viewing the output signals on a logic analyzer and then measuring the resulting adder delays. The results indicate that the TMR-RCA is the best approach for an FPGA fault tolerant implementation at bit widths up to 128 due to its simple design and the use of the fast carry chain. At higher bit widths of 256, the sparse Kogge-Stone adder using a graceful degradation approach proves to be superior to the TMR-RCA in terms of delay. The next chapter will summarize and conclude the thesis work that has been completed and discuss some possibilities for future work.

Chapter Five

Conclusions and Future Work

5.1 Conclusions

The fault tolerant adders implemented on FPGAs have been characterized with respect to their delay performance and logic complexity as a function of bit width. Basic fault tolerant adder designs like the Triple Modular Redundancy-Ripple Carry Adder (TMR-RCA) and error correcting regular Kogge-Stone adder were analyzed. A sparse Kogge-Stone adder which is fully fault tolerant in its lower half (i.e., in the ripple carry adders) was proposed. Simulation results demonstrate that this design is able to detect and correct errors in its RCA chains.

Architectures for implementing advanced fault tolerance techniques on the sparse Kogge-Stone adders were proposed. This includes the upper half fault tolerant sparse Kogge-Stone adder and a graceful degradation concept. Simulation and synthesis using FPGA design tools have validated the performance of the lower-half and upper-half sparse Kogge-Stone adders for both fault detection and correction. In this analysis, the TMR-RCA seems to be the most efficient approach for fault tolerant design on an FPGA in terms of its resources due to its simplicity and the ability to take the advantage of the fast-carry chain. However, for very large bit widths, there are indications that the Kogge-Stone adder offers superior performance over a ripple carry adder when implemented on an FPGA. A fault tolerant sparse Kogge-Stone adder is designed by taking advantage of the existing ripple carry adders in the architecture and adopting a similar approach to the TMR-RCA by inserting two additional ripple carry adders into the design. A graceful degradation approach is implemented with the sparse Kogge-Stone adder. In this approach, a faulty block is permanently replaced with a spare block. As the spare block is initially used for fault checking, the fault tolerant capability of the circuit is degraded in order to continue fault-free operation. The adder delay is faster for graceful degradation with an overhead of 1 ns from measured results and an overhead of 2 ns from the synthesis results independent of the bit widths when compared with the fault tolerant Kogge-Stone adder even though the resource utilization is similar.

5.2 Future Work

Two main areas for extending the present work are briefly considered. First, the development of methods and tools to make the proposed fault tolerant methods easier to implement can be undertaken. A method for easily scaling to larger bit widths for the upper half fault tolerant sparse Kogge-Stone adder should be investigated. Automated techniques for implementing the fully fault tolerant sparse Kogge-Stone adder should be developed. Second, a largely unexplored area of research is the application of error correcting codes to fault tolerant adder designs. In digital communications, an additional number of bits is added to a message to allow the detection and correction of corrupted bits during transmission. A similar method might be feasible with arithmetic circuits. An optimal error correcting code would take into account the logic structure of the adder and would enable fully fault tolerant implementations while adding a minimum amount of overhead.

References

- [1] L. Sterpone, M. SonzaReorda and M. Violante, "Evaluating Different Solutions to Design Fault Tolerant Sytems with SRAM-based FPGAs," *Journal of Electronic Testing: Theory and Applications*, vol. 23, pp. 47-54, 2007.
- [2] K. Kyriakoulakos and D. Pnevmatikatos, "A Novel SRAM-Based FPGA Architecture for Efficient TMR Fault Tolerance Support," *International Conference on Field Programmable Logic and Applications*, pp. 193-198, 2009.
- [3] L. Sterpone and M. Violanem, "Analysis of the Robustness of the TMR Architecture in SRAM-Based FPGAs", *IEEE Trans. Nucl. Sci.*, vol. 52, no. 5, pp. 1545-1549, Oct. 2005.
- [4] E. Stott, P. Sedcole, and P. Y.K. Cheung, "Fault Tolerant Methods for Reliability in FPGAs," *International Conference on Field Programmable Logic and Applications*, pp. 415-420, 2008.
- [5] S. Ghosh, P. Ndai and K. Roy, "A Novel Low Overhead Fault Tolerant Kogge-Stone Adder using Adaptive Clocking," *Design, Automation and Test*, pp. 366-371, 2008.
- [6] M. Abramovici, C. Stroud, C. Hamiltion, S. Wijesuriya, and V. Verma, "Using Roving STARs for On-Line Testing and Diagnosis of FPGAs in Fault-Tolerant Applications," *Test Conference*, pp. 973-982, 1999.
- [7] T. Lynch and E. E. Swartzlander, "A Spanning Tree Carry Lookahead Adder," *IEEE Transactions on Computers*, vol. 41, no. 8, pp. 931-939, Aug. 1992.
- [8] J. Vundavalli, "Design and Analysis of wide bit adders for FPGA Implementation," *MSEE Thesis, University of Texas at Tyler*, May 2010.

- [9] D. H. K. Hoe, C. Martinez, and J. Vundavalli, "Design and Characterization of Parallel Prefix Adders using FPGAs," *IEEE 43rd Southeastern Symposium on System Theory*, pp. 170-174, March 2011.
- [10] R. Iris, D. Hammerstrom, J. Harlow, W. H. Joyner Jr., C. Lau, D. Marculescu, A. Orailoglu, M. Pedram, "Architectures for Silicon Nanoelectronics and Beyond," *Computer*, vol. 40, no. 1, pp. 25-33, Jan. 2007.
- [11] N. Banerjee, C. Augustine, K. Roy, "Fault-Tolerance with Graceful Degradation in. Quality: A Design Methodology and its Application to Digital Signal Processing Systems," *IEEE International Symposium on, Defect and Fault Tolerance of VLSI Systems*, pp. 323-331, 1-3 Oct. 2008.
- [12] M. Abramovici, J. M. Emmert, and C. Stroud, "Roving STARs: An Integrated Approach to On-Line Testing, Diagnosis, and Fault Tolerance for FPGAs," *NASA/DoD Workshop on Evolvable Hardware*, pp. 73-92, 2001.
- [13] N. H. E. Weste and D. Harris, *CMOS VLSI Design*, Pearson-Addison-Wesley, Third edition, 2005.
- [14] J. M. Emmert, C. Stroud, and M. Abramovici, "Online Fault Tolerance for FPGA Logic Blocks," *IEEE Trans. on VLSI Systems*, vol. 15, no. 2, pp. 216-226, February 2007.

Appendices

Appendix: A

A1. VHDL Code for 32-bit TMR-RCA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TMR_RCA32 is

port(Cin : in std_logic;
      a,b : in std_logic_vector(32 downto 1);
      s: out std_logic_vector(32 downto 1);
      sum1,sum2,sum3: out std_logic_vector(32 downto 1);
      fault: in std_logic;
      cout : out std_logic);
end TMR_RCA32;

architecture Behavioral of TMR_RCA32 is

component adder1 is
port( Cin : in std_logic;
      A: in std_logic_vector(32 downto 1);
      B: in std_logic_vector(32 downto 1);
      S: inout std_logic_vector(32 downto 1);
      cout : out std_logic);
end component ;

component adder2 is
port( Cin : in std_logic;
      A: in std_logic_vector(32 downto 1);
      B: in std_logic_vector(32 downto 1);
      S: inout std_logic_vector(32 downto 1);
      cout : out std_logic);
end component ;

component adder3 is
port( Cin : in std_logic;
      fault: in std_logic;
      A: in std_logic_vector(32 downto 1);
      B: in std_logic_vector(32 downto 1);
```

Appendix: A (Continued)

```
S: inout std_logic_vector(32 downto 1);
cout : out std_logic);
end component ;

component comparator is
port(A: in std_logic_vector(32 downto 1);
     B: in std_logic_vector(32 downto 1);
     C: in std_logic_vector(32 downto 1);
     O: out std_logic_vector(32 downto 1)
    );
end component;

signal S1,S2,S3 : std_logic_vector(32 downto 1);
signal EQ : std_logic_vector(32 downto 1);

begin

RCAdder1 : adder1 port map( Cin,a,b,S1,Cout);
RCAdder2 : adder2 port map( Cin,a,b,S2,Cout);
RCAdder3 : adder3 port map( Cin,fault,a,b,S3,Cout);
Compare1 :comparator port map(S1,S2,S3,EQ);

s<= EQ;
sum1 <= S1;
sum2 <= S2;
sum3 <= S3;

end Behavioral;
```

A2. VHDL Code for adder1 in 32-bit TMR-RCA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder is
port( Cin : in std_logic;
```

Appendix: A (Continued)

```
A: in std_logic_vector(32 downto 1);
B: in std_logic_vector(32 downto 1);

S: inout std_logic_vector(32 downto 1);
Cout : out std_logic);
end adder1 ;

architecture Behavioral of adder1 is

signal SUM : std_logic_vector(33 downto 1);

begin
Cout <= SUM(33);
SUM <= ("0" & A) + ("0" & B) + cin;
S(32 downto 1) <= SUM(32 downto 1);
end Behavioral;
```

A3. VHDL Code for adder2 in 32-bit TMR-RCA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder2 is
port( Cin : in std_logic;
      A: in std_logic_vector(32 downto 1);
      B: in std_logic_vector(32 downto 1);
      S: inout std_logic_vector(32 downto 1);
      Cout : out std_logic);
end adder2 ;

architecture Behavioral of adder2 is
signal SUM : std_logic_vector(33 downto 1);

begin
Cout <= SUM(33);
```

Appendix: A (Continued)

```
SUM <= ("0" & A) + ("0" & B) + cin;  
S(32 downto 1) <= SUM(32 downto 1);  
end Behavioral;
```

A4. VHDL Code for adder3 in 32-bit TMR-RCA

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;  
  
entity a3 is  
  port( Cin : in std_logic;  
        A: in std_logic_vector(32 downto 1);  
        B: in std_logic_vector(32 downto 1);  
        S: inout std_logic_vector(32 downto 1);  
        fault : in std_logic;  
        Cout : out std_logic);  
end a3 ;  
architecture Behavioral of a3 is  
  signal SUM : std_logic_vector(33 downto 1);  
  
begin  
  Cout <= SUM(33);  
  SUM <= ("0" & A) + ("0" & B) + cin+fault;  
  S(32 downto 1) <= SUM(32 downto 1);  
end Behavioral;
```

A5. VHDL Code for comparator in 32-bit TMR-RCA

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
entity comparator is  
  port(  
    A: in std_logic_vector(32 downto 1);  
    B: in std_logic_vector(32 downto 1);  
    C: in std_logic_vector(32 downto 1);  
    O: out std_logic_vector(32 downto 1);
```

Appendix: A (Continued)

```
error,allerror : out std_logic);  
    end comparator ;
```

architecture Behavioral of comparator is

Appendix: A (Continued)

```
begin  
process(A,B,C)  
begin  
  
    if ((A=B) and (A=C)) then  
        O <= A;  
        error <= '0';  
        allerror<= '0';  
    elsif (A=C) then  
        O <=C;  
        error <= '1';  
        allerror<= '0';  
    elsif (A=B) then  
        O <= B;  
        error <= '1';  
        allerror<= '0';  
    elsif (B=C) then  
        O<=C;  
        error <= '1';  
        allerror <= '0';  
    else  
        O <=(others =>'X');  
        error <= '1';  
        allerror<= '1';  
  
    end if;  
end process;  
end Behavioral;
```

Appendix: B

B1. VHDL Code for 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity koggecorrecttest is

port(  reset,set,clk,control : in std_logic;
      fault : in std_logic;
      a,b: in std_logic_vector(8 downto 1);
      shift : out std_logic;
      c : inout std_logic_vector(9 downto 1);
      gclk : in std_logic;
      gclk1 : in std_logic;
      smux : out std_logic_vector(9 downto 1);
      s : out std_logic_vector(9 downto 1));

end koggecorrecttest;

architecture Behavioral of koggecorrecttest is

component CntlMuxs is
port (x,y: in std_logic;
      z : out std_logic;
      control: in std_logic);
end component;

component mux is
port(x,y: in std_logic_vector(1 downto 0);
      sel : in std_logic;
      z: out std_logic_vector(1 downto 0));
end component;

component d_ff is
port (clk,reset,set : in STD_LOGIC;
```

Appendix: B (Continued)

```
d: in std_logic_vector( 1 downto 0);  
q : out STD_LOGIC_vector(1 downto 0));  
end component;
```

```
component outd_ff is  
port (d,clk,reset,set : in STD_LOGIC;  
      q : out STD_LOGIC);  
end component;
```

```
component outmux is  
port(v,w,x,y: in std_logic;  
      sel: in std_logic;  
      z: out std_logic_vector(1 downto 0));  
end component;
```

```
component GPblock is  
port( a,b: in std_logic;  
      g,p: out std_logic);  
end component;
```

```
component blackcell is  
port( x1,y1,x2,y2: in std_logic;  
      x12,y12: out std_logic);  
end component;
```

```
component faultgraycell is  
port( x1,y1,x2,fault: in std_logic;  
      x12: out std_logic);  
end component;
```

```
component graycell is  
port( x1,y1,x2: in std_logic;  
      x12: out std_logic);  
end component;
```

```
component buffer1 is  
port( x1: in std_logic;
```

Appendix: B (Continued)

```
x2: out std_logic;
end component;

component Sum is
port(p,c : in std_logic;
      s : out std_logic);
end component;

signal p1,g1: std_logic_vector(2 downto 0);
signal p2,g2: std_logic_vector(2 downto 0);
signal p3,g3: std_logic_vector(3 downto 0);
signal p4,g4,p5,g5,p6,g6,p7,g7,p8,g8,p9,g9: std_logic_vector(3 downto 0);
signal mux1,mux2,mux3,mux4,mux5,mux6,mux7,mux8,mux9:std_logic_vector(1
downto 0);
signal dff1,dff2,dff3,dff4,dff5,dff6,dff7,dff8,dff9:std_logic_vector(1 downto 0);
signal
outmux1,outmux2,outmux3,outmux4,outmux5,outmux6,outmux7,outmux8,outmux9:
std_logic_vector(1 downto 0);

signal ground: std_logic := '0';
signal rightend: std_logic_vector(1 downto 0) := "10";
signal leftend: std_logic_vector(1 downto 0) := "10";
signal a1b1,a2b2,a3b3,a4b4,a5b5,a6b6,a7b7,a8b8 : std_logic_vector(1 downto 0);
signal sout,soutdff: std_logic_vector(9 downto 1);
signal o1,o2,shiftenable,sce,seclk,se: std_logic := '1';
signal temp_count : std_logic_vector(1 downto 0) := "00";

begin

a1b1 <= a(1) & b(1);
a2b2 <= a(2) & b(2);
a3b3 <= a(3) & b(3);
a4b4 <= a(4) & b(4);
a5b5 <= a(5) & b(5);
a6b6 <= a(6) & b(6);
a7b7 <= a(7) & b(7);
a8b8 <= a(8) & b(8);
```


Appendix: B (Continued)

```
multiplexer1 : mux port map(a1b1,rightend,se,mux1);
multiplexer2 : mux port map(a2b2,a1b1,se,mux2);
multiplexer3 : mux port map(a3b3,a2b2,se,mux3);
multiplexer4 : mux port map(a4b4,a3b3,se,mux4);
multiplexer5 : mux port map(a5b5,a4b4,se,mux5);
multiplexer6 : mux port map(a6b6,a5b5,se,mux6);
multiplexer7 : mux port map(a7b7,a6b6,se,mux7);
multiplexer8 : mux port map(a8b8,a7b7,se,mux8);
multiplexer9 : mux port map(leftend,a8b8,se,mux9);
```

```
dfflop1: d_ff port map(clk,reset,set,mux1,dff1);
dfflop2: d_ff port map(clk,reset,set,mux2,dff2);
dfflop3: d_ff port map(clk,reset,set,mux3,dff3);
dfflop4: d_ff port map(clk,reset,set,mux4,dff4);
dfflop5: d_ff port map(clk,reset,set,mux5,dff5);
dfflop6: d_ff port map(clk,reset,set,mux6,dff6);
dfflop7: d_ff port map(clk,reset,set,mux7,dff7);
dfflop8: d_ff port map(clk,reset,set,mux8,dff8);
dfflop9: d_ff port map(clk,reset,set,mux9,dff9);
```

```
GPblock1 : GPblock port map(dff1(1),dff1(0),g1(0),p1(0));
GPblock2 : GPblock port map(dff2(1),dff2(0),g2(0),p2(0));
GPblock3 : GPblock port map(dff3(1),dff3(0),g3(0),p3(0));
GPblock4 : GPblock port map(dff4(1),dff4(0),g4(0),p4(0));
GPblock5 : GPblock port map(dff5(1),dff5(0),g5(0),p5(0));
GPblock6 : GPblock port map(dff6(1),dff6(0),g6(0),p6(0));
GPblock7 : GPblock port map(dff7(1),dff7(0),g7(0),p7(0));
GPblock8 : GPblock port map(dff8(1),dff8(0),g8(0),p8(0));
GPblock9 : GPblock port map(dff9(1),dff9(0),g9(0),p9(0));
```

```
Blkcell0 : blackcell port map(g3(0),p3(0),g2(0),p2(0),g3(1),p3(1));
Blkcell1 : blackcell port map(g4(0),p4(0),g3(0),p3(0),g4(1),p4(1));
Blkcell2 : blackcell port map(g5(0),p5(0),g4(0),p4(0),g5(1),p5(1));
Blkcell3 : blackcell port map(g5(1),p5(1),g3(1),p3(1),g5(2),p5(2));
Blkcell4 : blackcell port map(g6(0),p6(0),g5(0),p5(0),g6(1),p6(1));
Blkcell5 : blackcell port map(g6(1),p6(1),g4(1),p4(1),g6(2),p6(2));
Blkcell6 : blackcell port map(g7(0),p7(0),g6(0),p6(0),g7(1),p7(1));
Blkcell7 : blackcell port map(g7(1),p7(1),g5(1),p5(1),g7(2),p7(2));
```

Appendix: B (Continued)

Blkcell8 : blackcell port map(g8(0),p8(0),g7(0),p7(0),g8(1),p8(1));
Blkcell9 : blackcell port map(g8(1),p8(1),g6(1),p6(1),g8(2),p8(2));
Blkcell10: blackcell port map(g9(0),p9(0),g8(0),p8(0),g9(1),p9(1));
Blkcell11: blackcell port map(g9(1),p9(1),g7(1),p7(1),g9(2),p9(2));

Graycell0 : graycell port map(g2(0),p2(0),g1(0),g2(1));
Graycell1 : faultgraycell port map(g3(1),p3(1),g1(1),fault,g3(2));
Graycell2 : graycell port map(g4(1),p4(1),g2(1),g4(2));
Graycell3 : faultgraycell port map(g5(2),p5(2),g1(1),fault,g5(3));
--Graycell3 : graycell port map(g5(2),p5(2),g1(1),g5(3));
Graycell4 : graycell port map(g6(2),p6(2),g2(2),g6(3));
--Graycell5 : graycell port map(g7(2),p7(2),g3(2),g7(3));
Graycell5 : faultgraycell port map(g7(2),p7(2),g3(2),fault,g7(3));
Graycell6 : graycell port map(g8(2),p8(2),g4(2),g8(3));
Graycell7 : faultgraycell port map(g9(2),p9(2),g5(2),fault,g9(3));

Buffer01 : buffer1 port map(g1(0),g1(1));
Buffer02 : buffer1 port map(g2(1),g2(2));
Buffer03 : buffer1 port map(g3(2),g3(3));
Buffer04 : buffer1 port map(g4(2),g4(3));

outmultiplexer1: outmux port map(p1(0),c(1),ground,ground,se,outmux1);
outmultiplexer2: outmux port map(p2(0),c(2),p1(0),c(1),se,outmux2);
outmultiplexer3: outmux port map(p3(0),c(3),p2(0),c(2),se,outmux3);
outmultiplexer4: outmux port map(p4(0),c(4),p3(0),c(3),se,outmux4);
outmultiplexer5: outmux port map(p5(0),c(5),p4(0),c(4),se,outmux5);
outmultiplexer6: outmux port map(p6(0),c(6),p5(0),c(5),se,outmux6);
outmultiplexer7: outmux port map(p7(0),c(7),p6(0),c(6),se,outmux7);
outmultiplexer8: outmux port map(p8(0),c(8),p7(0),c(7),se,outmux8);
outmultiplexer9: outmux port map(p9(0),c(9),p8(0),c(8),se,outmux9);

out_dff1 : outd_ff port map(sout(1),o1,reset,set,soutdff(1));
out_dff2 : outd_ff port map(sout(2),o2,reset,set,soutdff(2));
out_dff3 : outd_ff port map(sout(3),o1,reset,set,soutdff(3));
out_dff4 : outd_ff port map(sout(4),o2,reset,set,soutdff(4));
out_dff5 : outd_ff port map(sout(5),o1,reset,set,soutdff(5));
out_dff6 : outd_ff port map(sout(6),o2,reset,set,soutdff(6));
out_dff7 : outd_ff port map(sout(7),o1,reset,set,soutdff(7));
out_dff8 : outd_ff port map(sout(8),o2,reset,set,soutdff(8));

Appendix: B (Continued)

```
out_dff9 : outd_ff port map(sout(9),o1,reset,set,soutdff(9));
```

```
Sum1: Sum port map(outmux1(1),ground,sout(1));
Sum2: Sum port map(outmux2(1),outmux1(0),sout(2));
Sum3: Sum port map(outmux3(1),outmux2(0),sout(3));
Sum4: Sum port map(outmux4(1),outmux3(0),sout(4));
Sum5: Sum port map(outmux5(1),outmux4(0),sout(5));
Sum6: Sum port map(outmux6(1),outmux5(0),sout(6));
Sum7: Sum port map(outmux7(1),outmux6(0),sout(7));
Sum8: Sum port map(outmux8(1),outmux7(0),sout(8));
Sum9: Sum port map(outmux9(1),outmux8(0),sout(9));
```

```
smux <= sout;
s <= soutdff;
```

```
CntrlMux1 : CntlMuxs port map(gclk1,gclk,o1,control);
CntrlMux2 : CntlMuxs port map(gclk,gclk1,o2,control);
```

```
c(1) <= g1(1);
c(2) <= g2(2);
c(3) <= g3(3);
c(4) <= g4(3);
c(5) <= g5(3);
c(6) <= g6(3);
c(7) <= g7(3);
c(8) <= g8(3);
c(9) <= g9(3);
```

```
shift <= se;
```

```
shiftproc : process(clk)
begin
    if clk'event and clk = '0' then
        se <= not se;
    end if;
end process;
end Behavioral;
```

Appendix: B (Continued)

B2. VHDL Code for mux in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux is
port (x,y: in std_logic_vector(1 downto 0);
      z : out std_logic_vector(1 downto 0);
      sel: in std_logic);
end mux;

architecture Behavioral of mux is
constant delay: time :=100ns;
begin
mux_proc : process(x,y,sel)
variable temp : std_logic_vector(1 downto 0);
begin
case sel is
when '0'=> temp:=x;
when '1'=> temp:=y;
when others => temp :="XX";

end case;
z<= temp;
end process mux_proc;

end Behavioral;
```

B3. VHDL Code for GPblock in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity GPblock is
port( a,b: in std_logic;
      g,p: out std_logic);
```

Appendix: B (Continued)

```
end GPblock;

architecture Behavioral of GPblock is
begin

    g <= a and b;
    p <= a xor b;

end Behavioral;
```

B4. VHDL Code for blackcell in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity blackcell is
port( x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end blackcell;

architecture Behavioral of blackcell is

begin
    x12 <= x1 or (y1 and x2);
    y12 <= y1 and y2;

end Behavioral;
```

B5. VHDL Code for graycell in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity graycell is
```

Appendix: B (Continued)

```
port( x1,y1,x2: in std_logic;  
      x12: out std_logic);  
end graycell;
```

```
architecture Behavioral of graycell is  
begin  
x12 <= x1 or(y1 and x2);  
end Behavioral;
```

B6. VHDL Code for faultgraycell in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity faultgraycell is  
port( x1,y1,x2,fault: in std_logic;  
      x12: out std_logic);  
end faultgraycell;
```

```
architecture Behavioral of faultgraycell is
```

```
begin  
x12 <= not(fault) and(x1 or(y1 and x2));  
  
end Behavioral;
```

B7. VHDL Code for buffer1 in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity buffer1 is  
port( x1: in std_logic;  
      x2: out std_logic);  
end buffer1;
```

Appendix: B (Continued)

architecture Behavioral of buffer1 is

begin

x2 <= x1;

end Behavioral;

B8. VHDL Code for outmux in 8-bit Kogge-Stone Fault Correcting Adder

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

entity outmux is

port (v,w,x,y: in std_logic;

z : out std_logic_vector(1 downto 0);

sel: in std_logic);

end outmux;

architecture Behavioral of outmux is

begin

mux_proc : process(v,w,x,y,sel)

variable temp : std_logic_vector(1 downto 0);

begin

case sel is

when '1' => temp:=v&w;

when '0' => temp:=x&y;

when others => temp := "XX";

end case;

z<= temp;

end process mux_proc;

end Behavioral;

B9. VHDL Code for sum in 8-bit Kogge-Stone Fault Correcting Adder

library IEEE;

Appendix: B (Continued)

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity sum is
port( p,c: in std_logic;
      s : out std_logic);
end sum;
```

architecture Behavioral of sum is

```
begin
s <= (p xor c);
```

```
end Behavioral;
```

B10. VHDL Code for CntlMuxs in 8-bit Kogge-Stone Fault Correcting Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity CntlMuxs is
port (x,y: in std_logic;
      z : out std_logic;
      control: in std_logic);
end CntlMuxs;
```

architecture Behavioral of CntlMuxs is

```
begin
mux_proc : process(x,y,control)
variable temp : std_logic;
```

```
begin
case control is
when '1'=> temp:=x;
when others => temp:=y;
```


Appendix: B (Continued)

```
end case;  
z<= temp;  
end process mux_proc;  
  
end Behavioral;
```

Appendix: C

C1. VHDL Code for 32-bit Kogge-Stone Adder (Lower half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity KoggeStoneAdder_32 is
port( cin,clk,fault: in std_logic;
      countout : out std_logic_vector( 1 downto 0);
      c8,cx8,cx16,cx24,cx32,c32,c24,c16: inout std_logic;
      a,b: in std_logic_vector(32 downto 1);
      c: inout std_logic_vector(32 downto 1);
      sum: out std_logic_vector(32 downto 1);
      error,controlout: out std_logic);
end KoggeStoneAdder_32;

architecture Behavioral of KoggeStoneAdder_32 is

component GPblock is
port( a,b: in std_logic;
      g,p: out std_logic);
end component;

component blackcell is
port( x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;

component graycell is
port( x1,y1,x2: in std_logic;
      x12: out std_logic);
end component;

component buffer1 is
port( x1: in std_logic;
      x2: out std_logic);
end component;
```

Appendix: C (Continued)

```
component bitcounter1 is
Port ( clk : in std_logic;
      count_out : out std_logic_vector(1 downto 0));
end component;
```

```
component ConcatenationRCA is
port( c0 : in std_logic;
      A : in std_logic_vector(8 downto 1);
      B : in std_logic_vector(8 downto 1);
      S : out std_logic_vector(8 downto 1);
      cx : out std_logic);
end component;
```

```
component FaultyAdder1 is
port( c0,fault : in std_logic;
      A : in std_logic_vector(8 downto 1);
      B : in std_logic_vector(8 downto 1);
      S : out std_logic_vector(8 downto 1);
      cx : out std_logic);
end component;
```

```
component comparator1 is
port(adder1 : in std_logic_vector(8 downto 1);
      adder2 : in std_logic_vector(8 downto 1);
      adder3 : in std_logic_vector(8 downto 1);
      adderout : out std_logic_vector(8 downto 1);
      error : out std_logic;
      allerror : out std_logic);
end component;
```

```
signal p1,g1: std_logic_vector(2 downto 0);
signal p2,g2: std_logic_vector(3 downto 0);
signal p3,g3: std_logic_vector(3 downto 0);
signal p4,g4,p5,g5,p6,g6,p7,g7: std_logic_vector(4 downto 0);
signal p8,g8,p9,g9,p10,g10,p11,g11,p12,g12,p13,g13,p14,g14,p15,g15,p16,g16:
std_logic_vector(5 downto 0);
```

Appendix: C (Continued)

```
signal p17,g17,p18,g18,p19,g19,p20,g20,p21,g21,p22,g22,p23,g23,p24,g24,p25,g25:
std_logic_vector(5 downto 0);
signal p26,g26,p27,g27,g28,p28,g29,p29,g30,p30,g31,p31,g32,p32:
std_logic_vector(5 downto 0);
signal s: std_logic_vector(32 downto 1);
signal DMRO2,DMRB,DMRA: std_logic_vector(8 downto 1);
signal DMRO1: std_logic_vector(8 downto 1);
signal AUT,compout: std_logic_vector(8 downto 1);
signal DMRCI,cdummy2,cdummy1,ae,e,control : std_logic;
signal count: std_logic_vector( 1 downto 0);
```

begin

```
GPblock1 : GPblock port map(a(1),b(1),g1(0),p1(0));
GPblock2 : GPblock port map(a(2),b(2),g2(0),p2(0));
GPblock3 : GPblock port map(a(3),b(3),g3(0),p3(0));
GPblock4 : GPblock port map(a(4),b(4),g4(0),p4(0));
GPblock5 : GPblock port map(a(5),b(5),g5(0),p5(0));
GPblock6 : GPblock port map(a(6),b(6),g6(0),p6(0));
GPblock7 : GPblock port map(a(7),b(7),g7(0),p7(0));
GPblock8 : GPblock port map(a(8),b(8),g8(0),p8(0));
GPblock9 : GPblock port map(a(9),b(9),g9(0),p9(0));
GPblock10 : GPblock port map(a(10),b(10),g10(0),p10(0));
GPblock11 : GPblock port map(a(11),b(11),g11(0),p11(0));
GPblock12 : GPblock port map(a(12),b(12),g12(0),p12(0));
GPblock13 : GPblock port map(a(13),b(13),g13(0),p13(0));
GPblock14 : GPblock port map(a(14),b(14),g14(0),p14(0));
GPblock15 : GPblock port map(a(15),b(15),g15(0),p15(0));
GPblock16 : GPblock port map(a(16),b(16),g16(0),p16(0));
GPblock17 : GPblock port map(a(17),b(17),g17(0),p17(0));
GPblock18 : GPblock port map(a(18),b(18),g18(0),p18(0));
GPblock19 : GPblock port map(a(19),b(19),g19(0),p19(0));
GPblock20 : GPblock port map(a(20),b(20),g20(0),p20(0));
GPblock21 : GPblock port map(a(21),b(21),g21(0),p21(0));
GPblock22 : GPblock port map(a(22),b(22),g22(0),p22(0));
GPblock23 : GPblock port map(a(23),b(23),g23(0),p23(0));
GPblock24 : GPblock port map(a(24),b(24),g24(0),p24(0));
GPblock25 : GPblock port map(a(25),b(25),g25(0),p25(0));
```

Appendix: C (Continued)

GPblock26 : GPblock port map(a(26),b(26),g26(0),p26(0));
GPblock27 : GPblock port map(a(27),b(27),g27(0),p27(0));
GPblock28 : GPblock port map(a(28),b(28),g28(0),p28(0));
GPblock29 : GPblock port map(a(29),b(29),g29(0),p29(0));
GPblock30 : GPblock port map(a(30),b(30),g30(0),p30(0));
GPblock31 : GPblock port map(a(31),b(31),g31(0),p31(0));
GPblock32 : GPblock port map(a(32),b(32),g32(0),p32(0));

Blkcell0 : blackcell port map(g2(0),p2(0),g1(0),p1(0),g2(1),p2(1));
Blkcell1 : blackcell port map(g3(0),p3(0),g2(0),p2(0),g3(1),p3(1));
Blkcell2 : blackcell port map(g4(0),p4(0),g3(0),p3(0),g4(1),p4(1));
Blkcell3 : blackcell port map(g4(1),p4(1),g2(1),p2(1),g4(2),p4(2));
Blkcell4 : blackcell port map(g5(0),p5(0),g4(0),p4(0),g5(1),p5(1));
Blkcell5 : blackcell port map(g5(1),p5(1),g3(1),p3(1),g5(2),p5(2));
Blkcell6 : blackcell port map(g6(0),p6(0),g5(0),p5(0),g6(1),p6(1));
Blkcell7 : blackcell port map(g6(1),p6(1),g4(1),p4(1),g6(2),p6(2));
Blkcell8 : blackcell port map(g7(0),p7(0),g6(0),p6(0),g7(1),p7(1));
Blkcell9 : blackcell port map(g7(1),p7(1),g5(1),p5(1),g7(2),p7(2));
Blkcell10 : blackcell port map(g8(0),p8(0),g7(0),p7(0),g8(1),p8(1));
Blkcell11 : blackcell port map(g8(1),p8(1),g6(1),p6(1),g8(2),p8(2));
Blkcell12 : blackcell port map(g8(2),p8(2),g4(2),p4(2),g8(3),p8(3));
Blkcell13 : blackcell port map(g9(0),p9(0),g8(0),p8(0),g9(1),p9(1));
Blkcell14 : blackcell port map(g9(1),p9(1),g7(1),p7(1),g9(2),p9(2));
Blkcell15 : blackcell port map(g9(2),p9(2),g5(2),p5(2),g9(3),p9(3));
Blkcell16 : blackcell port map(g10(0),p10(0),g9(0),p9(0),g10(1),p10(1));
Blkcell17 : blackcell port map(g10(1),p10(1),g8(1),p8(1),g10(2),p10(2));
Blkcell18 : blackcell port map(g10(2),p10(2),g6(2),p6(2),g10(3),p10(3));
Blkcell19 : blackcell port map(g11(0),p11(0),g10(0),p10(0),g11(1),p11(1));
Blkcell20 : blackcell port map(g11(1),p11(1),g9(1),p9(1),g11(2),p11(2));
Blkcell21 : blackcell port map(g11(2),p11(2),g7(2),p7(2),g11(3),p11(3));
Blkcell22 : blackcell port map(g12(0),p12(0),g11(0),p11(0),g12(1),p12(1));
Blkcell23 : blackcell port map(g12(1),p12(1),g10(1),p10(1),g12(2),p12(2));
Blkcell24 : blackcell port map(g12(2),p12(2),g8(2),p8(2),g12(3),p12(3));
Blkcell25 : blackcell port map(g13(0),p13(0),g12(0),p12(0),g13(1),p13(1));
Blkcell26 : blackcell port map(g13(1),p13(1),g11(1),p11(1),g13(2),p13(2));
Blkcell27 : blackcell port map(g13(2),p13(2),g9(2),p9(2),g13(3),p13(3));
Blkcell28 : blackcell port map(g14(0),p14(0),g13(0),p13(0),g14(1),p14(1));
Blkcell29 : blackcell port map(g14(1),p14(1),g12(1),p12(1),g14(2),p14(2));

Appendix: C (Continued)

Blkcell30 : blackcell port map(g14(2),p14(2),g10(2),p10(2),g14(3),p14(3));
Blkcell31 : blackcell port map(g15(0),p15(0),g14(0),p14(0),g15(1),p15(1));
Blkcell32 : blackcell port map(g15(1),p15(1),g13(1),p13(1),g15(2),p15(2));
Blkcell33 : blackcell port map(g15(2),p15(2),g11(2),p11(2),g15(3),p15(3));
Blkcell34 : blackcell port map(g16(0),p16(0),g15(0),p15(0),g16(1),p16(1));
Blkcell35 : blackcell port map(g16(1),p16(1),g14(1),p14(1),g16(2),p16(2));
Blkcell36 : blackcell port map(g16(2),p16(2),g12(2),p12(2),g16(3),p16(3));
Blkcell37 : blackcell port map(g16(3),p16(3),g12(2),p12(2),g16(4),p16(4));
Blkcell38 : blackcell port map(g17(0),p17(0),g16(0),p16(0),g17(1),p17(1));
Blkcell39 : blackcell port map(g17(1),p17(1),g15(1),p15(1),g17(2),p17(2));
Blkcell40 : blackcell port map(g17(2),p17(2),g13(2),p13(2),g17(3),p17(3));
Blkcell41 : blackcell port map(g17(3),p17(3),g9(3),p9(3),g17(4),p17(4));
Blkcell42 : blackcell port map(g18(0),p18(0),g17(0),p17(0),g18(1),p18(1));
Blkcell43 : blackcell port map(g18(1),p18(1),g16(1),p16(1),g18(2),p18(2));
Blkcell44 : blackcell port map(g18(2),p18(2),g14(2),p14(2),g18(3),p18(3));
Blkcell45 : blackcell port map(g18(3),p18(3),g10(3),p10(3),g18(4),p18(4));
Blkcell46 : blackcell port map(g19(0),p19(0),g18(0),p18(0),g19(1),p19(1));
Blkcell47 : blackcell port map(g19(1),p19(1),g17(1),p17(1),g19(2),p19(2));
Blkcell48 : blackcell port map(g19(2),p19(2),g15(2),p15(2),g19(3),p19(3));
Blkcell49 : blackcell port map(g19(3),p19(3),g11(3),p11(3),g19(4),p19(4));
Blkcell50 : blackcell port map(g20(0),p20(0),g19(0),p19(0),g20(1),p20(1));
Blkcell51 : blackcell port map(g20(1),p20(1),g18(1),p18(1),g20(2),p20(2));
Blkcell52 : blackcell port map(g20(2),p20(2),g16(2),p16(2),g20(3),p20(3));
Blkcell53 : blackcell port map(g20(3),p20(3),g12(3),p12(3),g20(4),p20(4));
Blkcell54 : blackcell port map(g21(0),p21(0),g20(0),p20(0),g21(1),p21(1));
Blkcell55 : blackcell port map(g21(1),p21(1),g19(1),p19(1),g21(2),p21(2));
Blkcell56 : blackcell port map(g21(2),p21(2),g17(2),p17(2),g21(3),p21(3));
Blkcell57 : blackcell port map(g21(3),p21(3),g13(3),p13(3),g21(4),p21(4));
Blkcell58 : blackcell port map(g22(0),p22(0),g21(0),p21(0),g22(1),p22(1));
Blkcell59 : blackcell port map(g22(1),p22(1),g20(1),p20(1),g22(2),p22(2));
Blkcell60 : blackcell port map(g22(2),p22(2),g18(2),p18(2),g22(3),p22(3));
Blkcell61 : blackcell port map(g22(3),p22(3),g14(3),p14(3),g22(4),p22(4));
Blkcell62 : blackcell port map(g23(0),p23(0),g22(0),p22(0),g23(1),p23(1));
Blkcell63 : blackcell port map(g23(1),p23(1),g21(1),p21(1),g23(2),p23(2));
Blkcell64 : blackcell port map(g23(2),p23(2),g19(2),p19(2),g23(3),p23(3));
Blkcell65 : blackcell port map(g23(3),p23(3),g15(3),p15(3),g23(4),p23(4));
Blkcell66 : blackcell port map(g24(0),p24(0),g23(0),p23(0),g24(1),p24(1));
Blkcell67 : blackcell port map(g24(1),p24(1),g22(1),p22(1),g24(2),p24(2));

Appendix: C (Continued)

Blkcell68 : blackcell port map(g24(2),p24(2),g20(2),p20(2),g24(3),p24(3));
Blkcell69 : blackcell port map(g24(3),p24(3),g16(3),p16(3),g24(4),p24(4));
Blkcell70 : blackcell port map(g25(0),p25(0),g24(0),p24(0),g25(1),p25(1));
Blkcell71 : blackcell port map(g25(1),p25(1),g23(1),p23(1),g25(2),p25(2));
Blkcell72 : blackcell port map(g25(2),p25(2),g21(2),p21(2),g25(3),p25(3));
Blkcell73 : blackcell port map(g25(3),p25(3),g17(3),p17(3),g25(4),p25(4));
Blkcell74 : blackcell port map(g26(0),p26(0),g25(0),p25(0),g26(1),p26(1));
Blkcell75 : blackcell port map(g26(1),p26(1),g24(1),p24(1),g26(2),p26(2));
Blkcell76 : blackcell port map(g26(2),p26(2),g22(2),p22(2),g26(3),p26(3));
Blkcell77 : blackcell port map(g26(3),p26(3),g18(3),p18(3),g26(4),p26(4));
Blkcell78 : blackcell port map(g27(0),p27(0),g26(0),p26(0),g27(1),p27(1));
Blkcell79 : blackcell port map(g27(1),p27(1),g25(1),p25(1),g27(2),p27(2));
Blkcell80 : blackcell port map(g27(2),p27(2),g23(2),p23(2),g27(3),p27(3));
Blkcell81 : blackcell port map(g27(3),p27(3),g19(3),p19(3),g27(4),p27(4));
Blkcell82 : blackcell port map(g28(0),p28(0),g27(0),p27(0),g28(1),p28(1));
Blkcell83 : blackcell port map(g28(1),p28(1),g26(1),p26(1),g28(2),p28(2));
Blkcell84 : blackcell port map(g28(2),p28(2),g24(2),p24(2),g28(3),p28(3));
Blkcell85 : blackcell port map(g28(3),p28(3),g20(3),p20(3),g28(4),p28(4));
Blkcell86 : blackcell port map(g29(0),p29(0),g28(0),p28(0),g29(1),p29(1));
Blkcell87 : blackcell port map(g29(1),p29(1),g27(1),p27(1),g29(2),p29(2));
Blkcell88 : blackcell port map(g29(2),p29(2),g25(2),p25(2),g29(3),p29(3));
Blkcell89 : blackcell port map(g29(3),p29(3),g21(3),p21(3),g29(4),p29(4));
Blkcell90 : blackcell port map(g30(0),p30(0),g29(0),p29(0),g30(1),p30(1));
Blkcell91 : blackcell port map(g30(1),p30(1),g28(1),p28(1),g30(2),p30(2));
Blkcell92 : blackcell port map(g30(2),p30(2),g26(2),p26(2),g30(3),p30(3));
Blkcell93 : blackcell port map(g30(3),p30(3),g22(3),p22(3),g30(4),p30(4));
Blkcell94 : blackcell port map(g31(0),p31(0),g30(0),p30(0),g31(1),p31(1));
Blkcell95 : blackcell port map(g31(1),p31(1),g29(1),p29(1),g31(2),p31(2));
Blkcell96 : blackcell port map(g31(2),p31(2),g27(2),p27(2),g31(3),p31(3));
Blkcell97 : blackcell port map(g31(3),p31(3),g23(3),p23(3),g31(4),p31(4));

Graycell0 : graycell port map(g1(0),p1(0),cin,g1(1));
Graycell1 : graycell port map(g2(1),p2(1),cin,g2(2));
Graycell2 : graycell port map(g3(1),p3(1),g1(1),g3(2));
Graycell3 : graycell port map(g4(2),p4(2),cin,g4(3));
Graycell4 : graycell port map(g5(2),p5(2),g1(2),g5(3));
Graycell5 : graycell port map(g6(2),p6(2),g2(2),g6(3));
Graycell6 : graycell port map(g7(2),p7(2),g3(2),g7(3));

Appendix: C (Continued)

Graycell7 : graycell port map(g8(3),p8(3),cin,g8(4));
Graycell8 : graycell port map(g9(3),p9(3),g1(2),g9(4));
Graycell9 : graycell port map(g10(3),p10(3),g2(3),g10(4));
Graycell10 : graycell port map(g11(3),p11(3),g3(3),g11(4));
Graycell11 : graycell port map(g12(3),p12(3),g4(3),g12(4));
Graycell12 : graycell port map(g13(3),p13(3),g5(3),g13(4));
Graycell13 : graycell port map(g14(3),p14(3),g6(3),g14(4));
Graycell14 : graycell port map(g15(3),p15(3),g7(3),g15(4));
Graycell15 : graycell port map(g16(4),p16(4),cin,g16(5));
Graycell16 : graycell port map(g17(4),p17(4),g1(2),g17(5));
Graycell17 : graycell port map(g18(4),p18(4),g2(3),g18(5));
Graycell18 : graycell port map(g19(4),p19(4),g3(3),g19(5));
Graycell19 : graycell port map(g20(4),p20(4),g4(4),g20(5));
Graycell20 : graycell port map(g21(4),p21(4),g5(4),g21(5));
Graycell21 : graycell port map(g22(4),p22(4),g6(4),g22(5));
Graycell22 : graycell port map(g23(4),p23(4),g7(4),g23(5));
Graycell23 : graycell port map(g24(4),p24(4),g8(4),g24(5));
Graycell24 : graycell port map(g25(4),p25(4),g9(4),g25(5));
Graycell25 : graycell port map(g26(4),p26(4),g10(4),g26(5));
Graycell26 : graycell port map(g27(4),p27(4),g11(4),g27(5));
Graycell27 : graycell port map(g28(4),p28(4),g12(4),g28(5));
Graycell28 : graycell port map(g29(4),p29(4),g13(4),g29(5));
Graycell29 : graycell port map(g30(4),p30(4),g14(4),g30(5));
Graycell30 : graycell port map(g31(4),p31(4),g15(4),g31(5));
Graycell31 : graycell port map(g32(0),p32(0),c(31),g32(5));

Buffer01 : buffer1 port map(g1(1),g1(2));
Buffer02 : buffer1 port map(g2(2),g2(3));
Buffer03 : buffer1 port map(g3(2),g3(3));
Buffer04 : buffer1 port map(g4(3),g4(4));
Buffer05 : buffer1 port map(g5(3),g5(4));
Buffer06 : buffer1 port map(g6(3),g6(4));
Buffer07 : buffer1 port map(g7(3),g7(4));
Buffer08 : buffer1 port map(g8(4),g8(5));
Buffer09 : buffer1 port map(g9(4),g9(5));
Buffer010 : buffer1 port map(g10(4),g10(5));
Buffer011 : buffer1 port map(g11(4),g11(5));
Buffer012 : buffer1 port map(g12(4),g12(5));

Appendix: C (Continued)

```
Buffer013 : buffer1 port map(g13(4),g13(5));
Buffer014 : buffer1 port map(g14(4),g14(5));
Buffer015 : buffer1 port map(g15(4),g15(5));
c8 <= g8(4);
c16 <= g16(5);
c24 <= g24(5);
c32 <= g32(5);

CR1 : ConcatenationRCA port map(cin,a(8 downto 1),b(8 downto 1),s(8 downto
1),cx8);
CR9 : ConcatenationRCA port map(c8,a(16 downto 9),b(16 downto 9),s(16 downto
9),cx16);
CR17 : FaultyAdder1 port map(c16,fault,a(24 downto 17),b(24 downto 17),s(24
downto 17),cx24);
CR25 : ConcatenationRCA port map(c24,a(32 downto 25),b(32 downto 25),s(32
downto 25),cx32);

DMR1 : ConcatenationRCA port map(DMRCI,DMRA,DMRB,DMRO1,cdummy1);
DMR2 : ConcatenationRCA port map(DMRCI,DMRA,DMRB,DMRO2,cdummy2);
COMP2:Comparator1 port map(DMRO1,DMRO2,AUT,Compout,e,ae);
counter: bitcounter1 port map(control,count);

control <= (not e) and clk;
controlout<= control;
error<=e;
countout<= count;

IF_PRO: process(s,clk,count)
begin

if clk = '0' and (count = "00") then
    sum(32 downto 9) <= s(32 downto 9);
    sum(8 downto 1) <= compout;---perfect
elsif clk = '0' and (count = "01") then
    sum(32 downto 17) <= s(32 downto 17);
    sum(8 downto 1) <= s(8 downto 1);
    sum(16 downto 9) <= compout;--perfect
elsif clk = '0' and (count = "10") then
    sum(32 downto 25) <= s(32 downto 25);
```

Appendix: C (Continued)

```
    sum(8 downto 1) <= s(8 downto 1);
    sum(24 downto 17) <= compout;  --perfect
elsif clk = '0' and (count = "11") then
    sum(24 downto 1) <= s(24 downto 1);
    sum(32 downto 25) <= compout;--perfect
else
    sum <= s;
end if;
end process;
```

```
IF_PRO1: process(cin,c8,c16,c24,a,b,s,count)
```

```
begin
    if (count = "00") then
        DMRCI<= cin;
        DMRA <= A(8 downto 1);
        DMRB <= B(8 downto 1);
        AUT <= s(8 downto 1);

    elsif (count = "01") then
        DMRCI<= c8;
        DMRA <= A(16 downto 9);
        DMRB <= B(16 downto 9);
        AUT <= s(16 downto 9);

    elsif (count = "10") then
        DMRCI<= c16;
        DMRA <= A(24 downto 17);
        DMRB <= B(24 downto 17);
        AUT <= s(24 downto 17);

    elsif (count = "11") then
        DMRCI<= c24;
        DMRA <= A(32 downto 25);
        DMRB <= B(32 downto 25);
        AUT <= s(32 downto 25);
```

```
Else
```

Appendix: C (Continued)

```
DMRCI<= cin;
end if;
end process;
end Behavioral;
```

C2. VHDL Code for ConcatenationRCA in 32-bit Kogge-Stone Adder (Lower half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ConcatenationRCA is
port( c0 : in std_logic;
      A : in std_logic_vector(8 downto 1);
      B : in std_logic_vector(8 downto 1);
      S : out std_logic_vector(8 downto 1);
      cx : out std_logic);
end ConcatenationRCA;

architecture Behavioral of ConcatenationRCA is
signal SUM : std_logic_vector(9 downto 1);

begin
SUM <= ("0" & A) + ("0" & B) + c0;
cx <= SUM(9);

S(8) <= SUM(8) ;
S(7) <= SUM(7) ;
S(6) <= SUM(6) ;
S(5) <= SUM(5) ;
S(4) <= SUM(4) ;
S(3) <= SUM(3) ;
S(2) <= SUM(2) ;
S(1) <= SUM(1) ;

end Behavioral;
```

Appendix: C (Continued)

C3. VHDL Code for FaultyAdder1 in 32-bit Kogge-Stone Adder (Lower half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FaultyAdder1 is
port( c0,fault : in std_logic;
      A : in std_logic_vector(8 downto 1);
      B : in std_logic_vector(8 downto 1);
      S : out std_logic_vector(8 downto 1);
      cx : out std_logic);
end FaultyAdder1;

architecture Behavioral of FaultyAdder1 is
signal SUM : std_logic_vector(9 downto 1);

begin
SUM <= ("0" & A) + ("0" & B) + c0;

cx <= SUM(9);

S(8) <= SUM(8) ;
S(7) <= SUM(7)and (not fault);
S(6) <= SUM(6) ;
S(5) <= SUM(5) ;
S(4) <= SUM(4) ;
S(3) <= SUM(3) ;
S(2) <= SUM(2) ;
S(1) <= SUM(1) ;
end Behavioral;
```

C4. VHDL Code for comparator1 in 32-bit Kogge-Stone Adder (Lower half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Appendix: C (Continued)

entity comparator1 is

```
port (adder1 : in std_logic_vector(8 downto 1);
      adder2 : in std_logic_vector(8 downto 1);
      adder3 : in std_logic_vector(8 downto 1);
      adderout : out std_logic_vector(8 downto 1);
      error : out std_logic;
      allerror : out std_logic);
end comparator1;
```

architecture Behavioral of comparator1 is

begin

IF_PRO: process(adder1,adder2,adder3)

begin

if ((adder1 = adder2) and (adder1 = adder3)) then

adderout<= adder1;

error<= '0';

allerror<= '0';

elsif (adder1 = adder3) then

adderout<=adder1;

error<= '1';

allerror<= '0';

elsif (adder1=adder2) then

adderout <= adder1;

error<= '1';

allerror<= '0';

elsif (adder2= adder3) then

adderout <= adder2;

error<= '1';

allerror<= '0';

else

adderout<= (others => 'X');

error<= '1';

allerror<= '1';

end if;

Appendix: C (Continued)

```
end process;  
end Behavioral;
```

C5. VHDL Code for bitcounter1 in 32-bit Kogge-Stone Adder (Lower half)

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;  
  
entity bitcounter1 is  
Port ( clk : in std_logic;  
      count_out : out std_logic_vector(1 downto 0));  
end bitcounter1;  
  
architecture Behavioral of bitcounter1 is  
  
    signal temp_count : std_logic_vector(1 downto 0) := "00";  
begin  
    counting : process(clk,temp_count)  
    begin  
        if clk'event and clk = '1' then  
            temp_count <= temp_count + 1;  
        end if;  
        count_out <= temp_count;  
    end process;  
end Behavioral;
```

Appendix: D

D1. VHDL Code for 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity FTSparceUH32 is
port( cin,faultgreen,faultpurple,faultblue,fault2,clk : in std_logic;
      a,b: in std_logic_vector(32 downto 1);
      c32 : inout std_logic;
      -----TEST PORTS -----
      error,controlout,c8test,c16test,c24test : out std_logic;
      -----
      test : out std_logic_vector(8 downto 1);
      sum : out std_logic_vector(32 downto 1));
end FTSparceUH32;

architecture Behavioral of FTSparceUH32 is

component GPblock is
port( a,b: in std_logic;
      g,p: out std_logic);
end component;

component blackcell is
port( x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;

component graycell is
port( x1,y1,x2: in std_logic;
      x12: out std_logic);
end component;

component ConcatenationRCA is
port( c0 : in std_logic;
      a: in std_logic_vector(7 downto 0);
      b: in std_logic_vector(7 downto 0);
```

Appendix: D (Continued)

```
S: out std_logic_vector(7 downto 0);
cx: out std_logic);
end component;
component greengroup is
port( clk,RCcarry,fault,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7,x8,y8,cx: in
std_logic;
      x14,y14,cout: out std_logic);
end component;
component purplegroup is
port(
clk,RCcarry,fault,x10,y10,x11,y11,x12,y12,x13,y13,x14,y14,x15,y15,x16,y16,x17,y1
7,x9,y9,cx: in std_logic;
      x14x,y14x,cout: out std_logic);
end component;
component bluegroup is
port(
clk,RCcarry,fault,fault2,x17,y17,x18,y18,x19,y19,x20,y20,x21,y21,x22,y22,x23,y23,
x24,y24,x9,y9,cx: in std_logic;
      cout: out std_logic);
end component;

signal p0,g0: std_logic_vector(1 downto 0);
signal p1,g1: std_logic_vector(2 downto 0);
signal p2,g2: std_logic_vector(3 downto 0);
signal p3,g3: std_logic_vector(3 downto 0);
signal p4,g4,p5,g5,p6,g6,p7,g7: std_logic_vector(4 downto 0);
signal p8,g8,p9,g9,p10,g10,p11,g11,p12,g12,p13,g13,p14,g14,p15,g15,p16,g16:
std_logic_vector(5 downto 0);
signal p17,g17,p18,g18,p19,g19,p20,g20,p21,g21,p22,g22,p23,g23,p24,g24,p25,g25:
std_logic_vector(5 downto 0);
signal p26,g26,p27,g27,g28,p28,g29,p29,g30,p30,g31,p31: std_logic_vector(5
downto 0);
--2nd part
signal
g32,p32,p33,g33,p34,g34,p35,g35,p36,g36,p37,g37,p38,g38,p39,g39,p40,g40,p41,g4
1: std_logic_vector(6 downto 0);
signal p42,g42,p43,g43,p44,g44,p45,g45,p46,g46,p47,g47,p48,g48,p49,g49,p50,g50:
std_logic_vector(6 downto 0);
```


Appendix: D (Continued)

```
signal
p51,g51,p52,g52,p53,g53,p54,g54,p55,g55,p56,g56,p57,g57,p58,g58,p59,g59,p60,g6
0: std_logic_vector(6 downto 0);
signal p61,g61,p62,g62,p63,g63: std_logic_vector(6 downto 0);

signal s: std_logic_vector(32 downto 1);
signal DMRO2,DMRB,DMRA: std_logic_vector(8 downto 1);
signal DMRO1: std_logic_vector(8 downto 1);
signal AUT,compout: std_logic_vector(8 downto 1);
signal DMRCI,cdummy2,cdummy1,ae,e : std_logic;

signal count: std_logic_vector( 1 downto 0);

signal g64,g65,g66,g67,g68,g69,g70,g71,g72,g73,g74,g75,g76,g77,g78,g79:
std_logic_vector(7 downto 0);
signal p64,p65,p66,p67,p68,p69,p70,p71,p72,p73,p74,p75,p76,p77,p78,p79:
std_logic_vector(7 downto 0);
signal g80,g81,g82,g83,g84,g85,g86,g87,g88,g89,g90,g91,g92,g93,g94,g95:
std_logic_vector(7 downto 0);
signal p80,p81,p82,p83,p84,p85,p86,p87,p88,p89,p90,p91,p92,p93,p94,p95:
std_logic_vector(7 downto 0);
signal
g96,g97,g98,g99,g100,g101,g102,g103,g104,g105,g106,g107,g108,g109,g110,g111:
std_logic_vector(7 downto 0);
signal
p96,p97,p98,p99,p100,p101,p102,p103,p104,p105,p106,p107,p108,p109,p110,p111:
std_logic_vector(7 downto 0);
signal
g112,g113,g114,g115,g116,g117,g118,g119,g120,g121,g122,g123,g124,g125,g126,g
127: std_logic_vector(7 downto 0);
signal
p112,p113,p114,p115,p116,p117,p118,p119,p120,p121,p122,p123,p124,p125,p126,p
127: std_logic_vector(7 downto 0);

signal r1,y1,b1,pr1,pr2,gr1,gr2 : std_logic;
signal errorc8,errorc16,errorc24 : std_logic;
signal c8,c16,c24: std_logic;
signal cx8,cx16,cx24,cx32 : std_logic;
begin
```

Appendix: D (Continued)

GPblock1 : GPblock port map(a(1),b(1),g1(0),p1(0));
GPblock2 : GPblock port map(a(2),b(2),g2(0),p2(0));
GPblock3 : GPblock port map(a(3),b(3),g3(0),p3(0));
GPblock4 : GPblock port map(a(4),b(4),g4(0),p4(0));
GPblock5 : GPblock port map(a(5),b(5),g5(0),p5(0));
GPblock6 : GPblock port map(a(6),b(6),g6(0),p6(0));
GPblock7 : GPblock port map(a(7),b(7),g7(0),p7(0));
GPblock8 : GPblock port map(a(8),b(8),g8(0),p8(0));
GPblock9 : GPblock port map(a(9),b(9),g9(0),p9(0));
GPblock10 : GPblock port map(a(10),b(10),g10(0),p10(0));
GPblock11 : GPblock port map(a(11),b(11),g11(0),p11(0));
GPblock12 : GPblock port map(a(12),b(12),g12(0),p12(0));
GPblock13 : GPblock port map(a(13),b(13),g13(0),p13(0));
GPblock14 : GPblock port map(a(14),b(14),g14(0),p14(0));
GPblock15 : GPblock port map(a(15),b(15),g15(0),p15(0));
GPblock16 : GPblock port map(a(16),b(16),g16(0),p16(0));
GPblock17 : GPblock port map(a(17),b(17),g17(0),p17(0));
GPblock18 : GPblock port map(a(18),b(18),g18(0),p18(0));
GPblock19 : GPblock port map(a(19),b(19),g19(0),p19(0));
GPblock20 : GPblock port map(a(20),b(20),g20(0),p20(0));
GPblock21 : GPblock port map(a(21),b(21),g21(0),p21(0));
GPblock22 : GPblock port map(a(22),b(22),g22(0),p22(0));
GPblock23 : GPblock port map(a(23),b(23),g23(0),p23(0));
GPblock24 : GPblock port map(a(24),b(24),g24(0),p24(0));
GPblock25 : GPblock port map(a(25),b(25),g25(0),p25(0));
GPblock26 : GPblock port map(a(26),b(26),g26(0),p26(0));
GPblock27 : GPblock port map(a(27),b(27),g27(0),p27(0));
GPblock28 : GPblock port map(a(28),b(28),g28(0),p28(0));
GPblock29 : GPblock port map(a(29),b(29),g29(0),p29(0));
GPblock30 : GPblock port map(a(30),b(30),g30(0),p30(0));
GPblock31 : GPblock port map(a(31),b(31),g31(0),p31(0));
GPblock32 : GPblock port map(a(32),b(32),g32(0),p32(0));

green3 : greengroup port

map(clk,cx8,faultgreen,g8(0),p8(0),g7(0),p7(0),g6(0),p6(0),g5(0),p5(0),g4(0),p4(0),g3(0),p3(0),g2(0),p2(0),g1(0),p1(0),cin,gr1,gr2,c8);

Appendix: D (Continued)

purple3 : purplegroup port

```
map(clk,cx16,faultpurple,g16(0),p16(0),g15(0),p15(0),g14(0),p14(0),g13(0),p13(0),g12(0),p12(0),g11(0),p11(0),g10(0),p10(0),g9(0),p9(0),gr1,gr2,cin,pr1,pr2,c16);
```

blue3 : bluegroup port

```
map(clk,cx24,faultblue,fault2,g24(0),p24(0),g23(0),p23(0),g22(0),p22(0),g21(0),p21(0),g20(0),p20(0),g19(0),p19(0),g18(0),p18(0),g17(0),p17(0),pr1,pr2,c8,c24);
```

```
c8test<=errorc8;
```

```
c16test<=errorc16;
```

```
c24test<=errorc24;
```

```
CR1 : ConcatenationRCA port map(cin,a(8 downto 1),b(8 downto 1),s(8 downto 1),cx8);
```

```
CR2 : ConcatenationRCA port map(c8,a(16 downto 9),b(16 downto 9),s(16 downto 9),cx16);
```

```
CR3 : ConcatenationRCA port map(c16,a(24 downto 17),b(24 downto 17),s(24 downto 17),cx24);
```

```
CR4 : ConcatenationRCA port map(c24,a(32 downto 25),b(32 downto 25),s(32 downto 25),c32);
```

```
sum <=s;
```

```
end Behavioral;
```

D2. VHDL Code for greengroup in 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity greengroup is
```

```
port( clk,RCcarry,fault,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7,x8,y8,cx: in std_logic;
```

```
      x14,y14,cout: out std_logic);
```

```
end greengroup;
```

```
architecture Beavioral of greengroup is
```

```
component graycell is
```

```
port( x1,y1,x2: in std_logic;
```

```
      x12: out std_logic);
```

Appendix: D (Continued)

end component;

```
component faultygraycell is
port( fault,x1,y1,x2: in std_logic;
      x12: out std_logic);
end component;
```

```
component blackcell is
port( x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;
```

```
component faultyblackcell is
port( fault,x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;
```

```
signal g2,p2,g4,p4,g5,p5,g6,p6,g8,p8,g9,p9,g10,p10 : std_logic;
signal g2a,p2a,g4a,p4a,g5a,p5a,g6a,p6a,g8a,p8a,g9a,p9a,g10a,p10a : std_logic;
signal couta,cout1 : std_logic;
```

```
begin
Blkcell0 : blackcell port map(x1,y1,x2,y2,g2,p2);
Blkcell2 : blackcell port map(x3,y3,x4,x4,g4,p4);
Blkcell3 : blackcell port map(g4,p4,g2,p2,g6,p6);
Blkcell6 : blackcell port map(x5,y5,x6,y6,g5,p5);
Blkcell10 : blackcell port map(x7,y7,x8,y8,g8,p8);
Blkcell11 : blackcell port map(g8,p8,g5,p5,g9,p9);
Blkcell12 : blackcell port map(g9,p9,g6,p6,g10,p10);
Graycell7 : graycell port map(g10,p10,cx,cout1);
Blkcell0a : faultyblackcell port map(fault,x1,y1,x2,y2,g2a,p2a);
Blkcell2b : faultyblackcell port map(fault,x3,y3,x4,x4,g4a,p4a);
Blkcell3c : faultyblackcell port map(fault,g4a,p4a,g2a,p2a,g6a,p6a);
Blkcell6d : faultyblackcell port map(fault,x5,y5,x6,y6,g5a,p5a);
Blkcell10e : faultyblackcell port map(fault,x7,y7,x8,y8,g8a,p8a);
Blkcell11f : faultyblackcell port map(fault,g8a,p8a,g5a,p5a,g9a,p9a);
Blkcell12g : faultyblackcell port map(fault,g9a,p9a,g6a,p6a,g10a,p10a);
Graycell7h : faultygraycell port map(fault,g10a,p10a,cx,couta);
```

Appendix: D (Continued)

```
IF_PRO: process(RCcarry,g10,p10,g10a,p10a,clk,cout1,couta)
begin
    if clk = '0' and clk'event then
        if RCcarry = cout1 then
            x14<= g10;
            y14<= p10;
            cout<= cout1;
        else
            x14<= g10a;
            y14<= p10a;
            cout<=couta;
        end if;
    end if;
end process;
end Behavioral;
```

D3. VHDL Code for purplegroup in 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity purplegroup is
port(
    clk,RCcarry,fault,x10,y10,x11,y11,x12,y12,x13,y13,x14,y14,x15,y15,x16,y16,x17,y1
    7,x9,y9,cx: in std_logic;
    x14x,y14x,cout: out std_logic);
end purplegroup;

architecture Behavioral of purplegroup is

    component graycell is
    port( x1,y1,x2: in std_logic;
          x12: out std_logic);
    end component;

    component blackcell is
```

Appendix: D (Continued)

```
port( x1,y1,x2,y2: in std_logic;  
      x12,y12: out std_logic);  
end component;
```

```
component faultygraycell is  
port( fault,x1,y1,x2: in std_logic;  
      x12: out std_logic);  
end component;  
component faultyblackcell is  
port( fault,x1,y1,x2,y2: in std_logic;  
      x12,y12: out std_logic);  
end component;
```

```
signal g10,p10,g11,p11,g12,p12,g13,p13,g14,p14,g15,p15,g16,p16,g17,p17,g18,p18  
: std_logic;  
signal  
g10a,p10a,g11a,p11a,g12a,p12a,g13a,p13a,g14a,p14a,g15a,p15a,g16a,p16a,g17a,p17  
a,g18a,p18a : std_logic;  
signal couta,cout1 : std_logic;
```

```
begin  
--group1  
Blkcell16 : blackcell port map(x10,y10,x9,y9,g11,p11);  
Blkcell22 : blackcell port map(x11,y11,x12,y12,g12,p12);  
Blkcell23 : blackcell port map(g12,p12,g11,p11,g13,p13);  
Blkcell28 : blackcell port map(x13,y13,x14,y14,g14,p14);  
Blkcell34 : blackcell port map(x15,y15,x16,y16,g15,p15);  
Blkcell35 : blackcell port map(g15,p15,g14,p14,g16,p16);  
Blkcell36 : blackcell port map(g16,p16,g13,p13,g17,p17);  
Blkcell37 : blackcell port map(g17,p17,g10,p10,g18,p18);  
Graycell15 : graycell port map(g18,p18,cx,cout1);
```

```
--group2(backup)  
Blkcell16a : faultyblackcell port map(fault,x10,y10,x9,y9,g11a,p11a);  
Blkcell22a : faultyblackcell port map(fault,x11,y11,x12,y12,g12a,p12a);  
Blkcell23a : faultyblackcell port map(fault,g12a,p12a,g11a,p11a,g13a,p13a);  
Blkcell28a : faultyblackcell port map(fault,x13,y13,x14,y14,g14a,p14a);  
Blkcell34a : faultyblackcell port map(fault,x15,y15,x16,y16,g15a,p15a);
```

Appendix: D (Continued)

```
Blkcell35a : faultyblackcell port map(fault,g15a,p15a,g14a,p14a,g16a,p16a);
Blkcell36a : faultyblackcell port map(fault,g16a,p16a,g13a,p13a,g17a,p17a);
Blkcell37a : faultyblackcell port map(fault,g17a,p17a,g10a,p10a,g18a,p18a);
Graycell15a : faultygraycell port map(fault,g18a,p18a,cx,couta);
```

```
IF_PRO: process(RCcarry,clk,g18,p18,g18a,p18a,cout1,couta)
begin
    if clk = '0' and clk'event then
        if RCcarry = cout1 then
            x14x<= g18;
            y14x<= p18;
            cout<= cout1;
        else
            x14x<= g18a;
            y14x<= p18a;
            cout<=couta;
        end if;
    end if;
end process;

end Behavioral;
```

D4. VHDL Code for bluegroup in 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bluegroup is
port(
    clk,RCcarry,fault,fault2,x17,y17,x18,y18,x19,y19,x20,y20,x21,y21,x22,y22,x23,y23,
    x24,y24,x9,y9,cx: in std_logic;
    cout: out std_logic);
end bluegroup;

architecture Behavioral of bluegroup is
    component graycell is
    port( x1,y1,x2: in std_logic;
          x12: out std_logic);
```

Appendix: D (Continued)

end component;

```
component faultygraycell is
port( fault,x1,y1,x2: in std_logic;
      x12: out std_logic);
end component;
```

```
component blackcell is
port( x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;
```

```
component faultyblackcell is
port( fault,x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;
```

```
signal
g17,p17,g19,p19,g20,p20,g21,p21,g22,p22,g23,p23,g24,p24,g25,p25,g26,p26,couta,c
out1 : std_logic;
signal
g17a,p17a,g19a,p19a,g20a,p20a,g21a,p21a,g22a,p22a,g23a,p23a,g24a,p24a,g25a,p25
a,g26a,p26a : std_logic;
```

begin

```
Blkcell42 : blackcell port map(x17,y17,x18,y18,g19,p19);
Blkcell50 : blackcell port map(x19,y19,x20,y20,g20,p20);
Blkcell51 : blackcell port map(g19,p19,g20,p20,g21,p21);
Blkcell58 : blackcell port map(x21,y21,x22,y22,g22,p22);
Blkcell66 : blackcell port map(x23,y23,x24,y24,g23,p23);
Blkcell67 : blackcell port map(g23,p23,g22,p22,g24,p24);
Blkcell68 : blackcell port map(g24,p24,g25,p25,g25,p25);
Blkcell69 : blackcell port map(g25,p25,g17,p17,g26,p26);
Graycell11 : faultygraycell port map(fault2,g26,p26,cx,cout1);
```


Appendix: D (Continued)

```
Blkcell42a : faultyblackcell port map(x17,y17,x18,y18,g19a,p19a);
Blkcell50a : faultyblackcell port map(x19,y19,x20,y20,g20a,p20a);
Blkcell51a : faultyblackcell port map(g19a,p19a,g20a,p20a,g21a,p21a);
Blkcell58a : faultyblackcell port map(x21,y21,x22,y22,g22a,p22a);
Blkcell66a : faultyblackcell port map(x23,y23,x24,y24,g23a,p23a);
Blkcell67a : faultyblackcell port map(g23a,p23a,g22a,p22a,g24a,p24a);
Blkcell68a : faultyblackcell port map(g24a,p24a,g25a,p25a,g25a,p25a);
Blkcell69a : faultyblackcell port map(g25a,p25a,g17a,p17a,g26a,p26a);
Graycell11a : faultygraycell port map(fault2,g26a,p26a,cx,couta);
```

```
IF_PRO: process(RCcarry,clk,cout1,couta)
begin
    if clk = '0' and clk'event then
        if RCcarry = cout1 then
            cout<= cout1;
        else
            cout<=couta;
        end if;
    end if;
end if;
end process;
end Behavioral;
```

D5. VHDL Code for ConcatenationRCA in 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity ConcatenationRCA is
port( c0 : in std_logic;
      A : in std_logic_vector(8 downto 1);
      B : in std_logic_vector(8 downto 1);
      S : out std_logic_vector(8 downto 1);
      cx : out std_logic);
end ConcatenationRCA;
```

architecture Behavioral of ConcatenationRCA is

Appendix: D (Continued)

```
signal SUM : std_logic_vector(9 downto 1);
```

```
begin
SUM <= ("0" & A) + ("0" & B) + c0;
cx <= SUM(9);
S(8) <= SUM(8);
S(7) <= SUM(7);
S(6) <= SUM(6);
S(5) <= SUM(5);
S(4) <= SUM(4);
S(3) <= SUM(3);
S(2) <= SUM(2);
S(1) <= SUM(1);
end Behavioral;
```

D6. VHDL Code for faultygraycell in 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity faultygraycell is
port( fault,x1,y1,x2: in std_logic;
      x12: out std_logic);
end faultygraycell;

architecture Behavioral of faultygraycell is
begin
x12 <= (not fault) and (x1 or(y1 and x2));
end Behavioral;
```

D7. VHDL Code for faultblackcell in 32-bit Kogge-Stone Adder (Upper half)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity faultyblackcell is
port( fault,x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end faultyblackcell;
```

Appendix: D (Continued)

architecture Behavioral of faultyblackcell is

begin

x12 <=(not fault) and(x1 or (y1 and x2));

y12 <= (not fault) and (y1 and y2);

end Behavioral;

Appendix: E

E1. VHDL Code for 32-bit Graceful Degradation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sparsegd32 is
port( cin,clk,fault : in std_logic;
      a,b: in std_logic_vector(32 downto 1);
      c8,c16,c24,c32 : inout std_logic;
      -----TEST PORTS -----
      error, controlout : out std_logic;
      counterout,countertest : out std_logic_vector(2 downto 0);
      -----
      cx8,cx16,cx24,cx32 : inout std_logic;
      test : out std_logic_vector(8 downto 1);
      sum : out std_logic_vector(32 downto 1));
end sparsegd32;

architecture Behavioral of sparsegd32 is

component GPblock is
port( a,b: in std_logic;
      g,p: out std_logic);
end component;

component blackcell is
port( x1,y1,x2,y2: in std_logic;
      x12,y12: out std_logic);
end component;

component graycell is
port( x1,y1,x2: in std_logic;
      x12: out std_logic);
end component;

component buffer1 is
```

Appendix: E (Continued)

```
port( x1: in std_logic;  
      x2: out std_logic);  
end component;
```

```
component FaultyAdder is  
port( c0,fault : in std_logic;  
      A : in std_logic_vector(8 downto 1);  
      B : in std_logic_vector(8 downto 1);  
      S : out std_logic_vector(8 downto 1);  
      cx : out std_logic);  
end component;
```

```
component ConcatenationRCA is  
port( c0 : in std_logic;  
      a: in std_logic_vector(7 downto 0);  
      b: in std_logic_vector(7 downto 0);  
      S : out std_logic_vector(7 downto 0);  
      cx : out std_logic);  
end component;
```

```
component bitcounter is  
Port ( clk : in std_logic;  
      count_out : out std_logic_vector(2 downto 0));  
end component;
```

```
component comparator is  
port(adder1 : in std_logic_vector(8 downto 1);  
      adder2 : in std_logic_vector(8 downto 1);  
      adder3 : in std_logic_vector(8 downto 1);  
      adderout : out std_logic_vector(8 downto 1);  
      error : out std_logic;  
      allerror : out std_logic);  
end component;
```

```
component comparator2 is  
port(
```

Appendix: E (Continued)

```
SUT : in std_logic_vector(8 downto 1);  
error : out std_logic;  
end component;
```

```
signal s: std_logic_vector(32 downto 1);  
signal DMRO2,DMRB,DMRA: std_logic_vector(32 downto 1);  
signal DMRO1: std_logic_vector(8 downto 1);  
signal AUT,compout: std_logic_vector(8 downto 1);  
signal cdummy2,cdummy1,ae,e : std_logic;
```

```
signal DMRC4,DMRC3,DMRC2,DMRC1,DMRCI : std_logic;  
signal DMR1A,DMR1B, DMR2A,DMR2B ,DMR3A,DMR3B ,DMR4A,DMR4B  
,DMR5A,DMR5B : std_logic_vector(8 downto 1);  
signal cmux1,cmux2,cmux3,cmux4,cmux5 : std_logic;  
signal smux1,smux2,smux3,smux4,smux5,stest: std_logic_vector(8 downto 1);
```

```
signal control : std_logic;
```

```
signal count,count1: std_logic_vector( 2 downto 0);
```

```
signal p1,g1: std_logic_vector(2 downto 0);  
signal p2,g2: std_logic_vector(3 downto 0);  
signal p3,g3: std_logic_vector(3 downto 0);  
signal p4,g4,p5,g5,p6,g6,p7,g7: std_logic_vector(4 downto 0);  
signal p8,g8,p9,g9,p10,g10,p11,g11,p12,g12,p13,g13,p14,g14,p15,g15,p16,g16:  
std_logic_vector(5 downto 0);  
signal p17,g17,p18,g18,p19,g19,p20,g20,p21,g21,p22,g22,p23,g23,p24,g24,p25,g25:  
std_logic_vector(5 downto 0);  
signal p26,g26,p27,g27,g28,p28,g29,p29,g30,p30,g31,p31,g32,p32:  
std_logic_vector(5 downto 0);
```

```
begin
```

```
GPblock1 : GPblock port map(a(1),b(1),g1(0),p1(0));  
GPblock2 : GPblock port map(a(2),b(2),g2(0),p2(0));  
GPblock3 : GPblock port map(a(3),b(3),g3(0),p3(0));  
GPblock4 : GPblock port map(a(4),b(4),g4(0),p4(0));
```

Appendix: E (Continued)

GPblock5 : GPblock port map(a(5),b(5),g5(0),p5(0));
GPblock6 : GPblock port map(a(6),b(6),g6(0),p6(0));
GPblock7 : GPblock port map(a(7),b(7),g7(0),p7(0));
GPblock8 : GPblock port map(a(8),b(8),g8(0),p8(0));
GPblock9 : GPblock port map(a(9),b(9),g9(0),p9(0));
GPblock10 : GPblock port map(a(10),b(10),g10(0),p10(0));
GPblock11 : GPblock port map(a(11),b(11),g11(0),p11(0));
GPblock12 : GPblock port map(a(12),b(12),g12(0),p12(0));
GPblock13 : GPblock port map(a(13),b(13),g13(0),p13(0));
GPblock14 : GPblock port map(a(14),b(14),g14(0),p14(0));
GPblock15 : GPblock port map(a(15),b(15),g15(0),p15(0));
GPblock16 : GPblock port map(a(16),b(16),g16(0),p16(0));
GPblock17 : GPblock port map(a(17),b(17),g17(0),p17(0));
GPblock18 : GPblock port map(a(18),b(18),g18(0),p18(0));
GPblock19 : GPblock port map(a(19),b(19),g19(0),p19(0));
GPblock20 : GPblock port map(a(20),b(20),g20(0),p20(0));
GPblock21 : GPblock port map(a(21),b(21),g21(0),p21(0));
GPblock22 : GPblock port map(a(22),b(22),g22(0),p22(0));
GPblock23 : GPblock port map(a(23),b(23),g23(0),p23(0));
GPblock24 : GPblock port map(a(24),b(24),g24(0),p24(0));
GPblock25 : GPblock port map(a(25),b(25),g25(0),p25(0));
GPblock26 : GPblock port map(a(26),b(26),g26(0),p26(0));
GPblock27 : GPblock port map(a(27),b(27),g27(0),p27(0));
GPblock28 : GPblock port map(a(28),b(28),g28(0),p28(0));
GPblock29 : GPblock port map(a(29),b(29),g29(0),p29(0));
GPblock30 : GPblock port map(a(30),b(30),g30(0),p30(0));
GPblock31 : GPblock port map(a(31),b(31),g31(0),p31(0));
GPblock32 : GPblock port map(a(32),b(32),g32(0),p32(0));

Blkcell0 : blackcell port map(g2(0),p2(0),g1(0),p1(0),g2(1),p2(1));
Blkcell2 : blackcell port map(g4(0),p4(0),g3(0),p3(0),g4(1),p4(1));
Blkcell3 : blackcell port map(g4(1),p4(1),g2(1),p2(1),g4(2),p4(2));
Blkcell6 : blackcell port map(g6(0),p6(0),g5(0),p5(0),g6(1),p6(1));
Blkcell10 : blackcell port map(g8(0),p8(0),g7(0),p7(0),g8(1),p8(1));
Blkcell11 : blackcell port map(g8(1),p8(1),g6(1),p6(1),g8(2),p8(2));
Blkcell12 : blackcell port map(g8(2),p8(2),g4(2),p4(2),g8(3),p8(3));
Blkcell16 : blackcell port map(g10(0),p10(0),g9(0),p9(0),g10(1),p10(1));
Blkcell22 : blackcell port map(g12(0),p12(0),g11(0),p11(0),g12(1),p12(1));

Appendix: E (Continued)

Blkcell23 : blackcell port map(g12(1),p12(1),g10(1),p10(1),g12(2),p12(2));
Blkcell24 : blackcell port map(g12(2),p12(2),g8(2),p8(2),g12(3),p12(3));
Blkcell28 : blackcell port map(g14(0),p14(0),g13(0),p13(0),g14(1),p14(1));
Blkcell34 : blackcell port map(g16(0),p16(0),g15(0),p15(0),g16(1),p16(1));
Blkcell35 : blackcell port map(g16(1),p16(1),g14(1),p14(1),g16(2),p16(2));
Blkcell36 : blackcell port map(g16(2),p16(2),g12(2),p12(2),g16(3),p16(3));
Blkcell37 : blackcell port map(g16(3),p16(3),g12(2),p12(2),g16(4),p16(4));
Blkcell42 : blackcell port map(g18(0),p18(0),g17(0),p17(0),g18(1),p18(1));
Blkcell50 : blackcell port map(g20(0),p20(0),g19(0),p19(0),g20(1),p20(1));
Blkcell51 : blackcell port map(g20(1),p20(1),g18(1),p18(1),g20(2),p20(2));
Blkcell52 : blackcell port map(g20(2),p20(2),g16(2),p16(2),g20(3),p20(3));
Blkcell53 : blackcell port map(g20(3),p20(3),g12(3),p12(3),g20(4),p20(4));
Blkcell58 : blackcell port map(g22(0),p22(0),g21(0),p21(0),g22(1),p22(1));
Blkcell66 : blackcell port map(g24(0),p24(0),g23(0),p23(0),g24(1),p24(1));
Blkcell67 : blackcell port map(g24(1),p24(1),g22(1),p22(1),g24(2),p24(2));
Blkcell68 : blackcell port map(g24(2),p24(2),g20(2),p20(2),g24(3),p24(3));
Blkcell69 : blackcell port map(g24(3),p24(3),g16(3),p16(3),g24(4),p24(4));
Blkcell74 : blackcell port map(g26(0),p26(0),g25(0),p25(0),g26(1),p26(1));
Blkcell82 : blackcell port map(g28(0),p28(0),g27(0),p27(0),g28(1),p28(1));
Blkcell83 : blackcell port map(g28(1),p28(1),g26(1),p26(1),g28(2),p28(2));
Blkcell84 : blackcell port map(g28(2),p28(2),g24(2),p24(2),g28(3),p28(3));
Blkcell85 : blackcell port map(g28(3),p28(3),g20(3),p20(3),g28(4),p28(4));

Graycell3 : graycell port map(g4(2),p4(2),cin,g4(3));
Graycell7 : graycell port map(g8(3),p8(3),cin,g8(4));
Graycell11 : graycell port map(g12(3),p12(3),g4(3),g12(4));
Graycell15 : graycell port map(g16(4),p16(4),cin,g16(5));
Graycell19 : graycell port map(g20(4),p20(4),g4(4),g20(5));
Graycell23 : graycell port map(g24(4),p24(4),g8(4),g24(5));
Graycell27 : graycell port map(g28(4),p28(4),g12(4),g28(5));

c8 <= g8(4);
c16 <= g16(5);
c24 <= g24(5);

CR1 : ConcatenationRCA port map(DMRCl,DMR1A,DMR1B,smux1,cmux1);
CR2 : ConcatenationRCA port map(DMRC1,DMR2A,DMR2B,smux2,cmux2);
CR3 : ConcatenationRCA port map(DMRC2,DMR3A,DMR3B,smux3,cmux3);

Appendix: E (Continued)

CR4 : FaultyAdder port map(DMRC3,fault,DMR4A,DMR4B,smux4,cmux4);
CR5 : ConcatenationRCA port map(DMRC4,DMR5A,DMR5B,smux5,cmux5);

COMP1: comparator2 port map(stest,e);
counter: bitcounter port map(control,count);
counter1: bitcounter port map(clk,count1);

control <= clk and (not e);
controlout <= control;
error<= e;
counterout <= count;
countertest <=(count1);

IF_PRO: process(s,clk,count)

begin

if clk = '0' and (count = "000") then

sum(32 downto 25) <= smux4;
sum(24 downto 17) <= smux3;
sum(16 downto 9) <= smux2;
stest<= smux1;
sum(8 downto 1) <= smux5;
c32<=cmux4;

elsif clk = '0' and (count = "001") then

sum(32 downto 25) <= smux4;
sum(24 downto 17) <= smux3;
sum(16 downto 9) <= smux2;
stest<= smux1;
sum(8 downto 1) <= smux5;
c32<=cmux4;

elsif clk = '0' and (count = "010") then

sum(32 downto 25) <= smux4;
sum(24 downto 17) <= smux3;
sum(16 downto 9) <= smux5;
sum(8 downto 1) <= smux1;

Appendix: E (Continued)

```
stest<= smux2;
c32<=cmux4;

elsif clk = '0' and (count = "011") then

sum(32 downto 25) <= smux4;
sum(24 downto 17) <= smux3;
sum(16 downto 9) <= smux5;
sum(8 downto 1) <= smux1;
stest<= smux2;
c32<=cmux4;

elsif clk = '0' and (count = "100") then
sum(32 downto 25) <= smux4;
sum(24 downto 17) <= smux5;
sum(16 downto 9) <= smux2;
sum(8 downto 1) <= smux1;
stest<= smux3;
c32<=cmux4;

elsif clk = '0' and (count = "101") then

sum(32 downto 25) <= smux4;
sum(24 downto 17) <= smux5;
sum(16 downto 9) <= smux2;
sum(8 downto 1) <= smux1;
stest<= smux3;
c32<=cmux4;

elsif clk = '0' and (count = "110") then

sum(32 downto 25) <= smux5;
sum(24 downto 17) <= smux3;
sum(16 downto 9) <= smux2;
sum(8 downto 1) <= smux1;
stest<= smux4;
c32<=cmux5;

elsif clk = '0' and (count = "111") then
```

Appendix: E (Continued)

```
sum(32 downto 25) <= smux5;
sum(24 downto 17) <= smux3;
sum(16 downto 9) <= smux2;
sum(8 downto 1) <= smux1;
stest<= smux4;
c32<=cmux5;

end if;
end process;

IF_PRO1: process(cin,c8,c16,c24,a,b,s,count)
begin
    if (count = "000") then
        DMRCI<= '1';
        DMR1A <= "11111111";
        DMR1B <= "11111111";
        DMRC1<= c8;
        DMR2A <= a(16 downto 9);
        DMR2B <= b(16 downto 9);
        DMRC2<= c16;
        DMR3A <= a(24 downto 17);
        DMR3B <= b(24 downto 17);
        DMRC3<= c24;
        DMR4A <= a(32 downto 25);
        DMR4B <= b(32 downto 25);
        DMRC4<= cin;
        DMR5A <= a(8 downto 1);
        DMR5B <= b(8 downto 1);

    elsif (count = "001") then
        DMRCI<= '0';
        DMR1A <= "00000000";
        DMR1B <= "00000000";
        DMRC1<= c8;
        DMR2A <= a(16 downto 9);
        DMR2B <= b(16 downto 9);
        DMRC2<= c16;
        DMR3A <= a(24 downto 17);
        DMR3B <= b(24 downto 17);
```

Appendix: E (Continued)

```
DMRC3<= c24;
DMR4A <= a(32 downto 25);
DMR4B <= b(32 downto 25);
DMRC4<= cin;
DMR5A <= a(8 downto 1);
DMR5B <= b(8 downto 1);

elsif (count = "010") then
    DMRCI<= cin;
    DMR1A <=a(8 downto 1);
    DMR1B <=b(8 downto 1);
    DMRC1<= '1';
    DMR2A <= "11111111";
    DMR2B <= "11111111";
    DMRC2<= c16;
    DMR3A <= a(24 downto 17);
    DMR3B <= b(24 downto 17);
    DMRC3<= c24;
    DMR4A <= a(32 downto 25);
    DMR4B <= b(32 downto 25);
    DMRC4<= c8;
    DMR5A <= a(16 downto 9);
    DMR5B <= b(16 downto 9);

elsif (count = "011") then
    DMRCI<= cin;
    DMR1A <=a(8 downto 1);
    DMR1B <=b(8 downto 1);
    DMRC1<= '0';
    DMR2A <= "00000000";
    DMR2B <= "00000000";
    DMRC2<= c16;
    DMR3A <= a(24 downto 17);
    DMR3B <= b(24 downto 17);
    DMRC3<= c24;
    DMR4A <= a(32 downto 25);
    DMR4B <= b(32 downto 25);
    DMRC4<= c8;
    DMR5A <= a(16 downto 9);
```

Appendix: E (Continued)

```
    DMR5B <= b(16 downto 9);
elsif (count = "100") then
    DMRCI<= cin;
    DMR1A <=a(8 downto 1);
    DMR1B <=b(8 downto 1);
    DMRC1<= c8;
    DMR2A <= a(16 downto 9);
    DMR2B <= b(16 downto 9);
    DMRC2<= '1';
    DMR3A <= "11111111";
    DMR3B <= "11111111";
    DMRC3<= c24;
    DMR4A <= a(32 downto 25);
    DMR4B <= b(32 downto 25);
    DMRC4<= c16;
    DMR5A <= a(24 downto 17);
    DMR5B <= b(24 downto 17);

elsif (count = "101") then
    DMRCI<= cin;
    DMR1A <=a(8 downto 1);
    DMR1B <=b(8 downto 1);
    DMRC1<= c8;
    DMR2A <= a(16 downto 9);
    DMR2B <= b(16 downto 9);
    DMRC2<= '0';
    DMR3A <= "00000000";
    DMR3B <= "00000000";
    DMRC3<= c24;
    DMR4A <= a(32 downto 25);
    DMR4B <= b(32 downto 25);
    DMRC4<= c16;
    DMR5A <= a(24 downto 17);
    DMR5B <= b(24 downto 17);
elsif (count = "110") then
    DMRCI<= cin;
    DMR1A <=a(8 downto 1);
    DMR1B <=b(8 downto 1);
```

Appendix: E (Continued)

```
DMRC1<= c8;
DMR2A <= a(16 downto 9);
DMR2B <= b(16 downto 9);
DMRC2<= c16;
DMR3A <= a(24 downto 17);
DMR3B <= b(24 downto 17);
DMRC3<= '1';
DMR4A <= "11111111";
DMR4B <= "11111111";
DMRC2<= c24;
DMR3A <= a(32 downto 25);
DMR3B <= b(32 downto 25);
else
  DMRCI<= cin;
  DMR1A <=a(8 downto 1);
  DMR1B <=b(8 downto 1);
  DMRC1<= c8;
  DMR2A <= a(16 downto 9);
  DMR2B <= b(16 downto 9);
  DMRC2<= c16;
  DMR3A <= a(24 downto 17);
  DMR3B <= b(24 downto 17);
  DMRC3<= '0';
  DMR4A <= "00000000";
  DMR4B <= "00000000";
  DMRC2<= c24;
  DMR3A <= a(32 downto 25);
  DMR3B <= b(32 downto 25);

  end if;
end process;
end Behavioral;
```

E2. VHDL Code for comparator2 in 32-bit Graceful Degradation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Appendix: E (Continued)

```
entity comparator2 is
port (SUT : in std_logic_vector(8 downto 1);
      error : out std_logic);
end comparator2;

architecture Behavioral of comparator2 is
begin
IF_PRO: process(SUT)
begin
    if ((SUT = "11111111") or (SUT = "00000000")) then

        error<= '0';
    else
        error <= '1';
    end if;
end process;
end Behavioral;
```

Appendix: F

F1. VHDL code for 32-bit TMR-RCA Implemented on Hardware

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TMRblockrom32 is
port (Clk: in std_logic;
      SEL: in STD_LOGIC;
      muxout: out std_logic_vector (31 downto 0);
      carrymux: out std_logic;
      count: out std_logic_vector(2 downto 0);
      Clkout: out std_logic);
end TMRblockrom32 ;

architecture Behavioral of TMRblockrom32 is
component TMR_RCA32 is
port(Cin : in std_logic;
      a,b : in std_logic_vector(32 downto 1);
      s: out std_logic_vector(32 downto 1);
      sum1,sum2,sum3: out std_logic_vector(32 downto 1);
      fault: in std_logic;
      cout : out std_logic);
end component;

----Rom Initialization----

component rom is
PORT ( clka : IN STD_LOGIC;
```


Appendix: F (Continued)

```
addra : IN STD_LOGIC_VECTOR(2 DOWNTO 0);

douta : OUT STD_LOGIC_VECTOR(64 DOWNTO 0) );

end component;

-----

component bitcounter IS

    port (clk: IN std_logic;

          count_out: out std_logic_VECTOR(2 downto 0));

end component;

signal a1 : std_logic_vector(31 downto 0);

signal b1 : std_logic_vector(31 downto 0) := (others => '0');

signal s1,s2,s3,s4 : std_logic_vector(31 downto 0);

signal cout : std_logic;

signal ci,fault : std_logic := '0';

signal address : std_logic_vector(2 downto 0);

signal data : std_logic_vector(64 downto 0);

begin

Clkout <= Clk;

b1 <= data(31 downto 0);

a1 <= data(63 downto 32);

ci <= data(64);

count <= address;

counter: bitcounter port map(Clk,address);

-----port mapping of rom-----

corerom: rom port map(Clk,address,data);
```

Appendix: F (Continued)

adderundertest: TMR_RCA32 port map(ci,a1,b1,s1,s2,s3,s4,fault,cout);

IF_PRO: process(SEL,a1,s1,ci,cout)

begin

if (SEL = '1') then

 muxout <= s1;
 carrymux <= cout;

else

 muxout <= a1;
 carrymux <= ci;

end if;

end process;

end Behavioral;

F2. VHDL code for 32-bit Sparse Kogge-Stone Lower Half Implemented on Hardware

library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sparceblockrom32 is

port (Clk: in std_logic;

 SEL: in STD_LOGIC;

 count: out std_logic_vector (2 downto 0);

 addercount: out std_logic_vector (1 downto 0);

 muxout: out std_logic_vector (31 downto 0);

 carrymux: out std_logic;

 Clkout: out std_logic);

end sparceblockrom32 ;

architecture Behavioral of sparceblockrom32 is

component KoggeStoneAdder_32 is

port(cin,clk,fault: in std_logic;

Appendix: F (Continued)

```
countout : out std_logic_vector( 1 downto 0);
c8,cx8,cx16,cx24,cx32,c32,c24,c16: inout std_logic;
a,b: in std_logic_vector(32 downto 1);
c: inout std_logic_vector(32 downto 1);
sum: out std_logic_vector(32 downto 1);
error,controlout: out std_logic);
end component;

----Rom Initialization----
component rom is
  PORT (clka : IN STD_LOGIC;
        addra : IN STD_LOGIC_VECTOR(2 DOWNT0 0);
        douta : OUT STD_LOGIC_VECTOR(64 DOWNT0 0)
  );
end component;

-----
component bitcounter IS
  port (clk: IN std_logic;
        count_out: out std_logic_VECTOR(2 downto 0));
end component;

signal a1 : std_logic_vector(32 downto 1);
signal b1,ci1 : std_logic_vector(32 downto 1) := (others => '0');
signal sum1 : std_logic_vector(32 downto 1);
signal error1,controlout1 : std_logic;
signal ci,fault1,c81,cx81,cx161,cx241,cx321,c321,c241,c161 : std_logic := '0';
signal address : std_logic_vector(2 downto 0);
signal coutout1 : std_logic_vector(1 downto 0);
signal data : std_logic_vector(64 downto 0);

begin
  Clkout <= Clk;
  b1 <= data(31 downto 0);
  a1 <= data(63 downto 32);
  ci <= data(64);
  count <= address;

  counter: bitcounter port map(Clk,address);
  -----port mapping of rom-----
```

Appendix: F (Continued)

```
corerom: rom port map(Clk,address,data);
```

```
-----  
adderundertest: KoggeStoneAdder_32 port  
map(ci,Clk,fault1,addercount,c81,cx81,cx161,cx241,cx321,c321,c241,c161,a1,b1,ci1,su  
m1,error1,controlout1);  
IF_PRO: process(SEL,a1,sum1,ci,controlout1)  
begin  
    if (SEL = '1') then  
        muxout <= sum1;  
        carrymux <= c321;  
    else  
        muxout <= a1;  
        carrymux <= ci;  
    end if;  
end process;  
end Behavioral;
```

F3. VHDL code for 32-bit Graceful Degradation Implemented on Hardware

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_ARITH.ALL;  
use IEEE.STD_LOGIC_UNSIGNED.ALL;  
  
entity blockrom32sparse is  
port (Clk: in std_logic;  
      SEL: in STD_LOGIC;  
      muxout: out std_logic_vector (31 downto 0);  
      carrymux: out std_logic;  
      count: out std_logic_vector(2 downto 0);  
      Clkout: out std_logic);  
end blockrom32sparse ;  
  
architecture Behavioral of blockrom32sparse is  
  
    component sparsegd32 is  
    port( cin,clk,fault : in std_logic;  
          a,b: in std_logic_vector(32 downto 1);
```

Appendix: F (Continued)

```
c8,c16,c24,c32 : inout std_logic;
-----TEST PORTS -----
error, controlout : out std_logic;
counterout,countertest : out std_logic_vector(2 downto 0);
-----

cx8,cx16,cx24,cx32 : inout std_logic;
test : out std_logic_vector(8 downto 1);
sum : out std_logic_vector(32 downto 1));
end component;

component rom is
  PORT (clka : IN STD_LOGIC;
        addra : IN STD_LOGIC_VECTOR(2 DOWNT0 0);
        douta : OUT STD_LOGIC_VECTOR(64 DOWNT0 0)
  );
end component;

component bitcounter IS
  port (clk: IN std_logic;
        count_out: out std_logic_VECTOR(2 downto 0));
end component;

signal a1 : std_logic_vector(31 downto 0);
signal b1 : std_logic_vector(31 downto 0) := (others => '0');
signal s : std_logic_vector(31 downto 0);
signal c1,c2,c3,c5,c8,c16,c24,c32 : std_logic;
signal ci,fault,error,controlout : std_logic := '0';
signal address,counterout,countertest : std_logic_vector(2 downto 0);
signal test: std_logic_vector(8 downto 1);
signal data : std_logic_vector(64 downto 0);

begin
  Clkout <= Clk;
  b1 <= data(31 downto 0);
  a1 <= data(63 downto 32);
  ci <= data(64);
  count<= address;

  counter: bitcounter port map(Clk,address);
```

Appendix: F (Continued)

-----port mapping of rom-----

corerom: rom port map(Clk,address,data);

adderundertest: sparsegd32 port

map(ci,clk,fault,a1,b1,c8,c16,c24,c32,error,controlout,counterout,countertest,c1,c2,c3,c5,test,s);

IF_PRO: process(SEL,a1,s,ci,c32)

begin

if (SEL = '1') then

muxout <= s;

carrymux <= c32;

else

muxout <= a1;

carrymux <= ci;

end if;

end process;

end Behavioral;

Appendix G

G.1 Other Fault Combinations for Upper Half Fault Tolerant Sparse Kogge-Stone Adder

Consider the cycle with index = 6 shown in following figures. The chosen inputs are $c_{in} = 1$, $a = \text{'ffffff'}$, and $b = \text{'ffffff'}$. In this case multiple fault combinations of fault are injected. The obtained output sum is correctly computed as 'ffffff' for all the combinations.

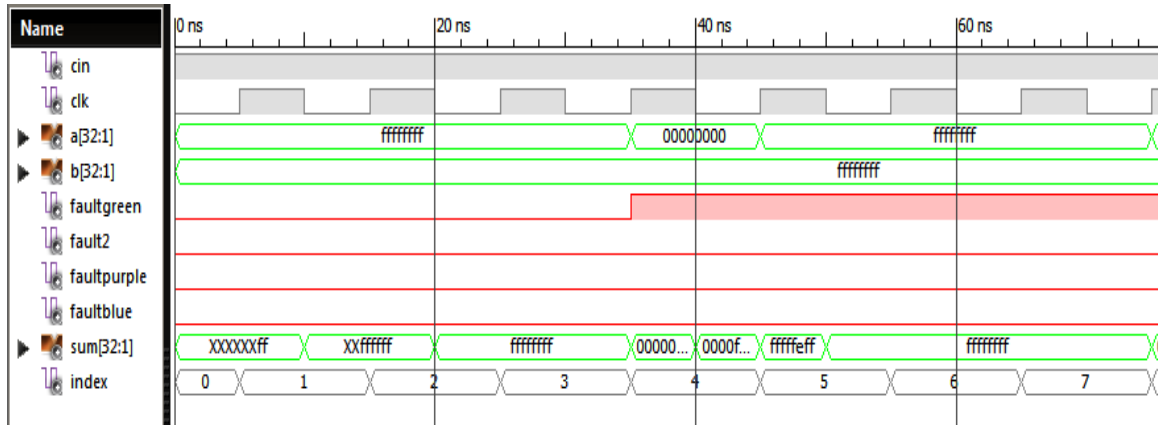


Figure G.1: Fault introduced in green section

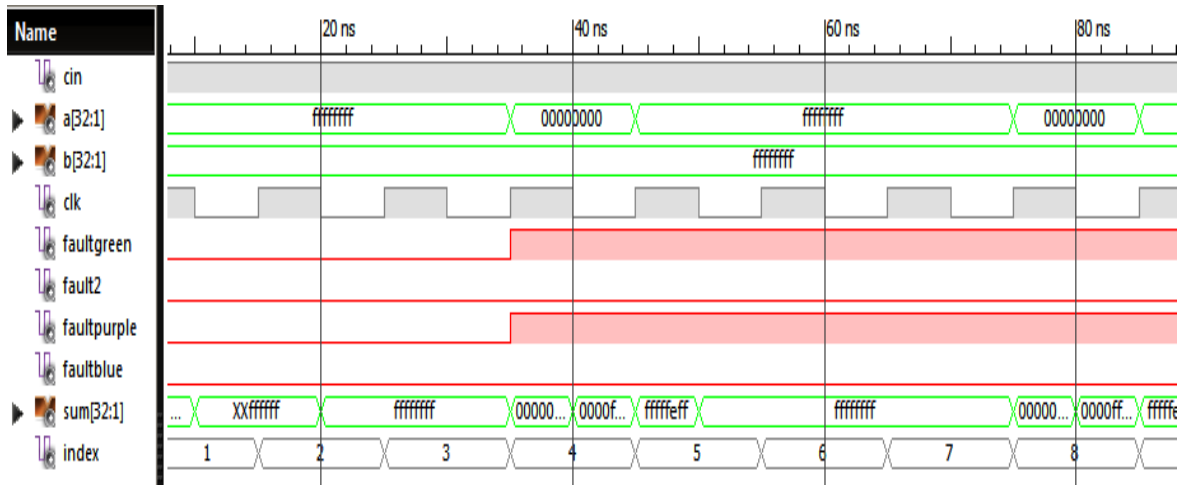


Figure G.2: Fault introduced in green and purple section

Appendix G (Continued)

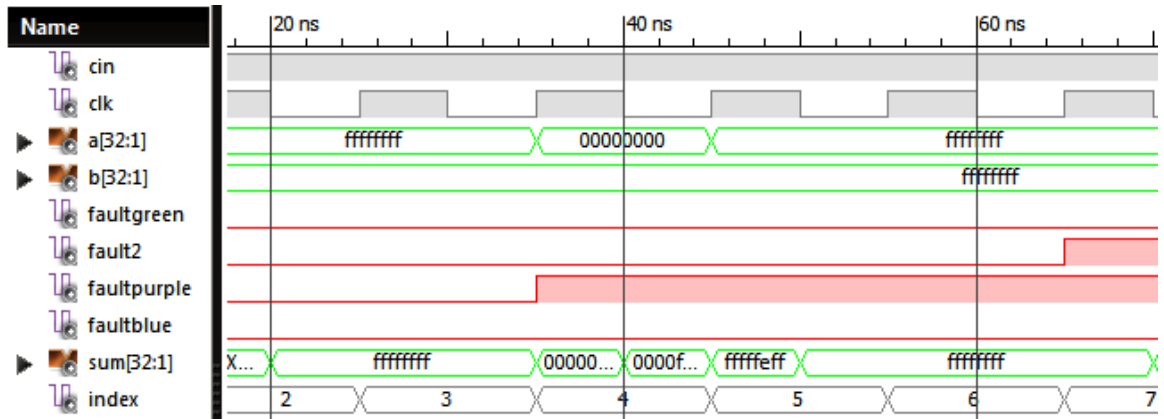


Figure G.3: Fault introduced in purple section

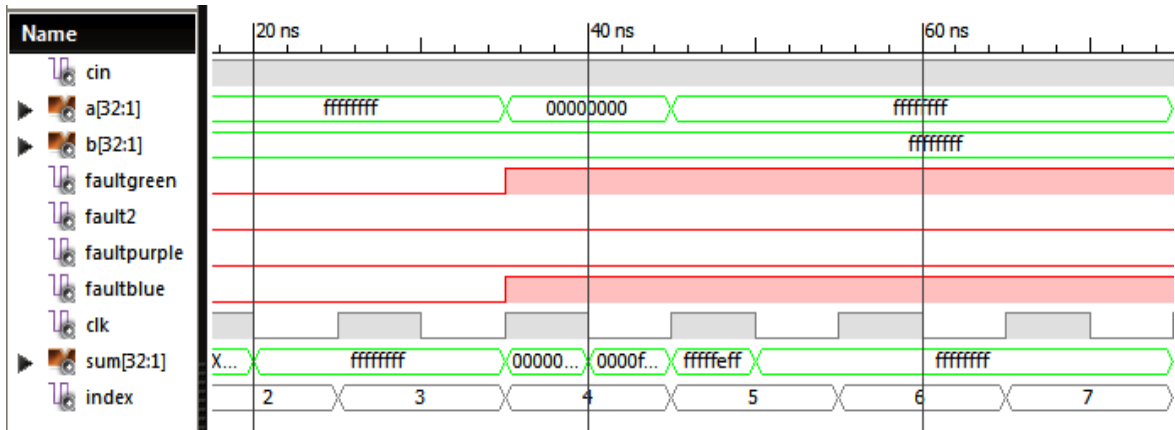


Figure G.4: Fault introduced in green and blue section

Appendix H

H1 Spartan-3E

The Spartan-3E starter kit is a complete development board that gives instant access to the platform capabilities of Spartan-3E family. Some of the specific features of the Spartan-3E FPGA are parallel NOR flash configuration, multiboot FPGA confirmation from parallel NOR flash PROM and it has SPI serial flash configuration. It also demonstrates the basic capabilities of micro blaze embedded processor and the Xilinx Embedded development Kit (EDK). It features a Xilinx platform Flash, USB and JTAG parallel programming interfaces with numerous FPGA configuration options. It is also compatible with MicroBlaze Embedded Development Kit (EDK) and PicoBlaze from Xilinx.

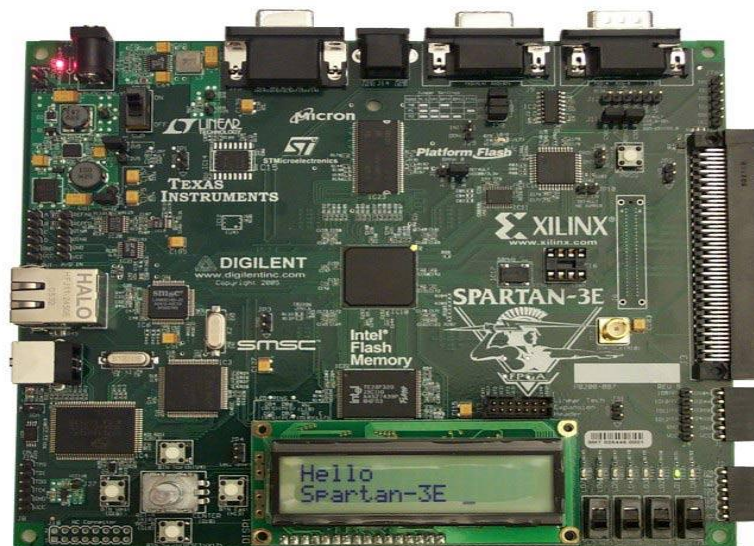


Figure H-1: Spartan-3E FPGA

Few of the key components and features of Spartan-3E are it has up to 18,624 look up tables (LUTs), 232 user I/O pins, 320 pin FPGA package and has over 10,000 logic cells.

Appendix: H (Continued)

One of the major applications of Spartan-3E is general prototyping. Its major markets include consumer, telecom/datacom, servers and storage.

H2 Virtex-5

The Virtex-5 FPGA series was introduced by Xilinx in 2006, which provides the newest most powerful features in the FPGA market. The Virtex series includes embedded fixed function hardware that is commonly used in functions such as memories, multipliers and microprocessor cores. It has 69,120 look up tables (LUTs), 20 total I/O banks and 640 maximum user I/O. It is the most advanced, high performance, optimal-utilization, FPGA fabric using real 6-input look-up table (LUT) technology. It uses the second generation Advanced Silicon Modular Block (ASML) column-based architecture. They are the world's first 65nm FPGA family fabricated in 1.0V, triple-oxide process technology. It contains five distinct platforms, each consisting of different features that address the needs of wide variety of advanced logic designs as mentioned in [14]. The five platforms of Virtex-5 are LX, LXT, SXT, TXT and FXT which include high speed serial connectivity.

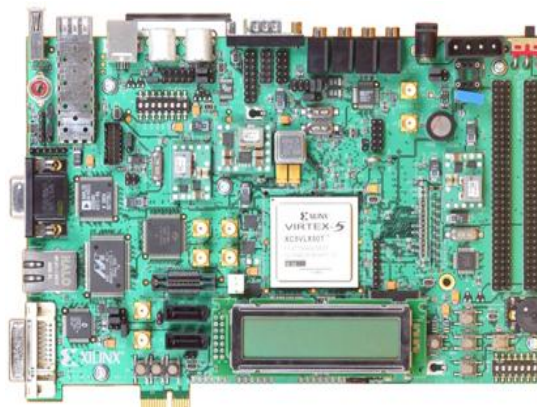


Figure H-2: Virtex-5 FPGA

Appendix: H (Continued)

Few of the Virtex-5 FPGA's vast applications include industrial, scientific, medical, telecom and networking, aerospace and defence, audio, video, broadcast, servers and storage, embedded and DSP and wireless infrastructure.

Appendix I

I1. Delay Calculation of TMR_RCA on Logic Analyzer

Aim: To obtain the total adder delay of the TMR_RCA of 16 bit using TLA 7012 Logic Analyzer.

Procedure:

The functionality of the adder to be tested is coded in VHDL and is verified using ISIM. The Xilinx ISE 12.4 software is used to synthesize the designs onto the Spartan 3E FPGA. A memory block called block rom is created to allow the arbitrary patterns of inputs which are applied to the adder design.

Step-1:

When the select pin N17 is high then the adder which is under test is included as shown in the Figure I.1. Thus the multiplexer select signal at each adder output decides whether to include the adder in the measured results or not. Consider the output obtained in this case i.e. when select N17 pin is high as output signal 1.

Where,

$$\text{Output signal 1} = \text{ROM} + \text{Adder under Test} + \text{Multiplexer} + X$$

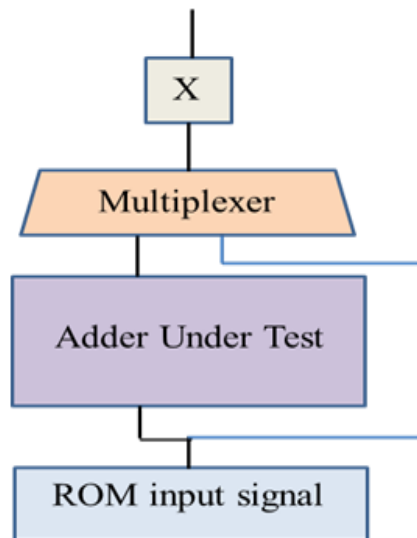


Figure I-1: Adder delay including ROM and multiplexer

Appendix I (Continued)



= non adder routing and input and output buffering

Then note all the reading for sum and carry of the adder accordingly based on the bit width when N17 pin is high.

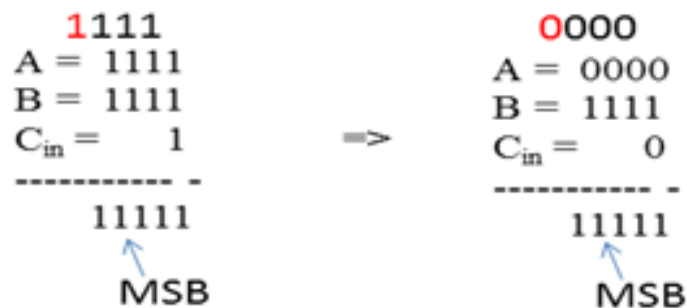
The following Table I.1 represents the input pattern used for obtaining the critical delay of the 16 bit TMR_RCA and the noted delays for sum and carry when select pin is high.

Table I-1: Input pattern chosen for testing TMR-RCA

C _{in}	Input A	Input B	Sum	ΔSum	ΔCarry
1	0000	FFFF	0000	7.383	9.161
0	0000	FFFF	FFFF	8.123	8.125
1	0000	FFFF	0000	7.363	9.141
0	0000	FFFF	FFFF	8.106	8.106
1	0000	FFFF	0000	7.383	9.161
0	0000	FFFF	FFFF	8.122	8.125
1	0000	FFFF	0000	7.364	9.141
0	FFFF	0000	FFFF	8.125	8.086

The highlighted values in the Table I-1 are the worst cases of sum and carry. The worst case for carry is at transition 8 to transition 1 and the worst case for sum is at the transition 7 to transition 8.

Explanation for the chosen Pattern for TMR-RCA:



Appendix I (Continued)

Here as seen above '1' is rippling all the way through carryout when all the inputs are high and even with the high carry-in.

'0' is rippling all the way through carryout when one of the inputs is high and another is low with a low carry-in.

Thus at this transition there will be worst case delay. Thus the pattern shown in Table I.1 is used while calculating the critical delay of the adder.

The highlighted bits in red are the obtained carryout for both the cases.

Note: The difference in the delays for sum and carry is due to the structure of BROM and routing.

This can be explained as follows. Figure I.2 shows a simplified view of Spartan-3 FPGA Carry and Arithmetic Logic in one logic cell.

Let t_{XOR} is the time delay due to XOR gate, t_{Buffer} is the time delay due to buffer and t_{Mux} is the time delay due to multiplexer

In general we know that $t_{XOR} > t_{Mux}$, but because of the buffer at the output of the multiplexer some additional delay of the buffer will be included apart from the delay of the multiplexer.

Thus $t_{XOR} < t_{Mux} + t_{Buffer}$

Here $t_{Mux} + t_{Buffer}$ is the carryout delay whereas t_{XOR} is the delay of the last sum i.e. sum_{16}

The simplified view of Spartan-3 FPGA Carry and Arithmetic Logic in one logic cell is shown below.

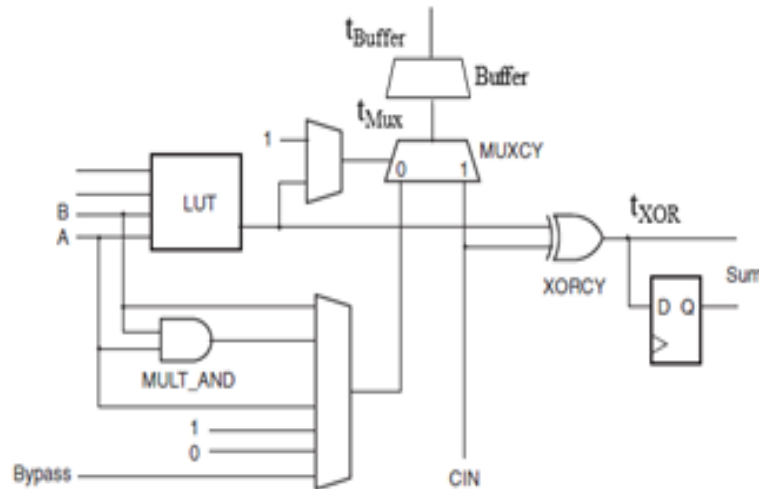


Figure I-2: Logic cell of Spartan 3E

Appendix I (Continued)

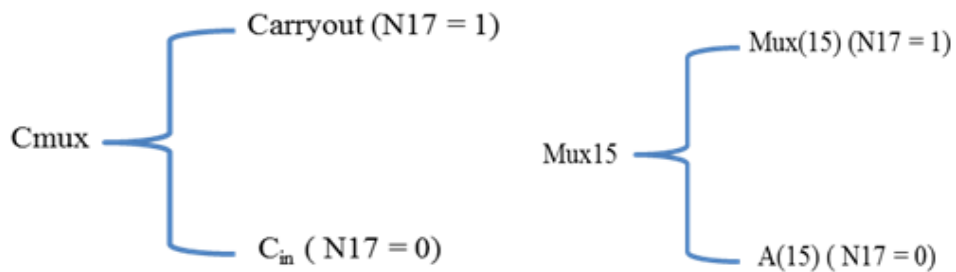
Thus our observed readings make sense showing carry delay is greater than the sum delay. Then note their corresponding worst case delay at select pin N17 is high for both sum and carry at different patterns of the corresponding worst case . Table I-2 shows the readings obtained.

Table I-2: Worst case carry and sum delays

	Mux(15)	Carryout
	8.125	9.18
	8.125	9.14
	8.145	9.16
	8.125	9.16
Average	8.13	9.16

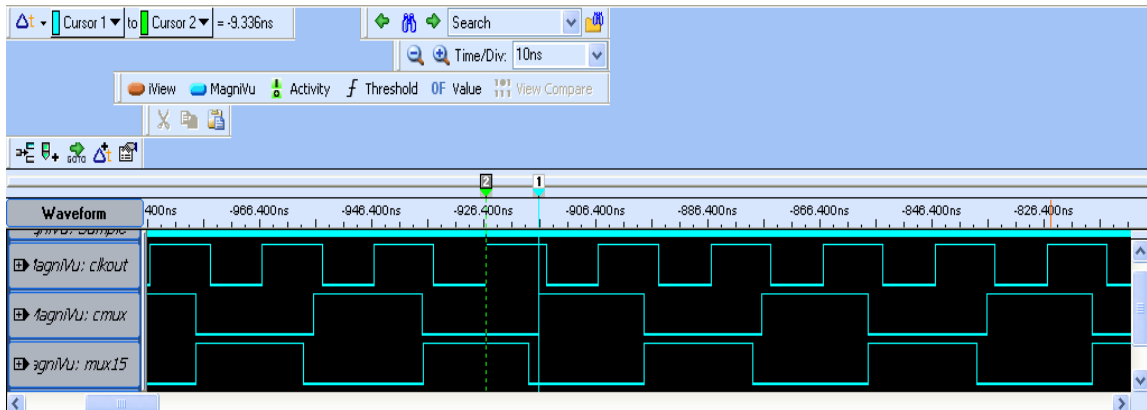
Note: Here Mux(15) represents Sum(15)
Thus Output signal 1 in this case is **9.16 ns.**

Brief discussion of selection with pin N17 is as shown below

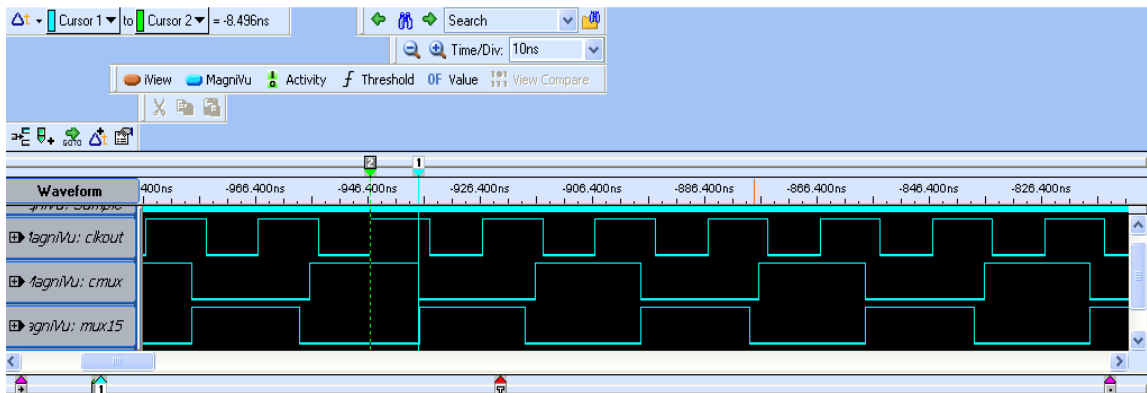


Here are some of the screen shots from the logic analyzer
For delay at Δ Carryout

Appendix I (Continued)



For delay at ΔSum_{16}



Step-2:

Now observe the delays of the inputs and the carry when the select pin N17 is low.

When the select pin N17 is low then the adder which is under test is excluded as shown in the Figure I.3. Consider the output obtained in this case i.e. when select N17 pin is low as output signal 2.

Where,

$$\text{Output signal 2} = \text{ROM} + \text{Multiplexer} + X$$

Appendix I (Continued)

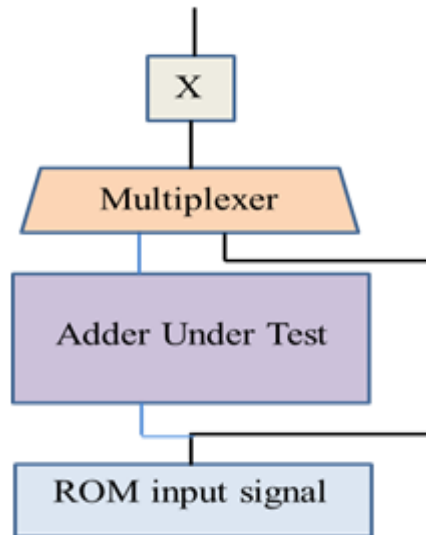


Figure I-3: Adder delay excluding ROM and multiplexer

Now note down its corresponding inputs and carry for different clock cycles as shown in the Table I.3 when select N17 is low

Table I-3: Worst case input delays

	Mux(15)	C _{in}
	3.77	4.336
	3.796	4.316
	3.769	4.317
	3.789	4.316
Average	3.781	4.321

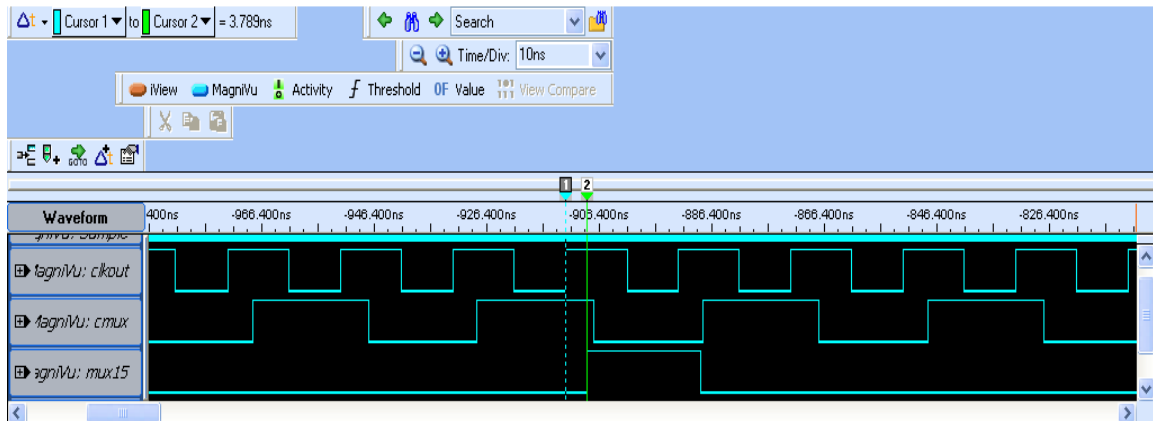
Thus Output signal 2 in this case is 4.321 ns.

Here Mux(15) will be input A(15) in Table I-3 when select pin is low.

Here is the screen shot taken while finding out the sum delay when the select pin is low.

For delay at ΔA_{16}

Appendix I (Continued)



Step-3: Critical Adder Delay

The adder delay is the difference between the average delay taken for the inputs to pass from the bitcounter to mux including the adder under test and excluding the adder under test i.e. result obtained at step 1 – result obtained at Step 2, gives the critical adder delay.

Therefore the critical adder delay for 16 bit sparse kogge is given as

Adder Delay = Output signal 1 – Output signal 2

$$= 9.16 - 4.321 = 4.838 \text{ ns.}$$

I2. Observing Worst Case Transition for Kogge Stone Adder

Objective:

The main objective is to observe the worst case pattern for Kogge-Stone adder which will be helpful in obtaining the critical delay of the Kogge stone adder.

Introduction:

First the functionality of the Kogge-Stone adder of 16bit is verified using ISIM. Then in order to test the critical adder delay, a memory block called ROM was instantiated on the FPGA using the core generator. The selection of the pattern of the inputs for the Kogge-Stone adder was discussed in the explanation given below.

Appendix I (Continued)

For the parallel prefix adders, only a specific pattern could be used for finding out the worst case delay. Thus by the structure of the Generate-Propagate blocks, a scheme was developed which considers the following subset of the input values to the GP blocks.

TableI-4: Subset of (g, p) relations used for Testing

(g_L, p_L)	(g_R, p_R)	$(g_L + p_L g_L, p_L p_R)$
(0,1)	(0,1)	(0,1)
(0,1)	(1,0)	(1,0)
(1,0)	(0,1)	(1,0)
(1,0)	(1,0)	(1,0)

The assigned (g, p) ordered pairs are $(1,0) = \text{True}$ and $(0,1) = \text{False}$. Hence the above table forms an OR Truth table. Thus if both the inputs of the GP block are false, then the output is false where as if both the inputs of the GP block are true, then the output is true. Hence an input pattern that alternates the (g, p) pairs of $(1,0)$ and $(0,1)$ will force the GP pair block to its alternate stages. The GP block pairs which are also fed by the predecessors will be alternating the states.

Thus the scheme will ensure the worst case delay in the parallel prefix adder since every block is active. The procedure below explains the chosen pattern for obtaining the worst case delay of the Kogge-Stone adder.

Procedure:

Consider a Kogge-Stone adder of 3bit which is shown in Figure I-4

Appendix I (Continued)

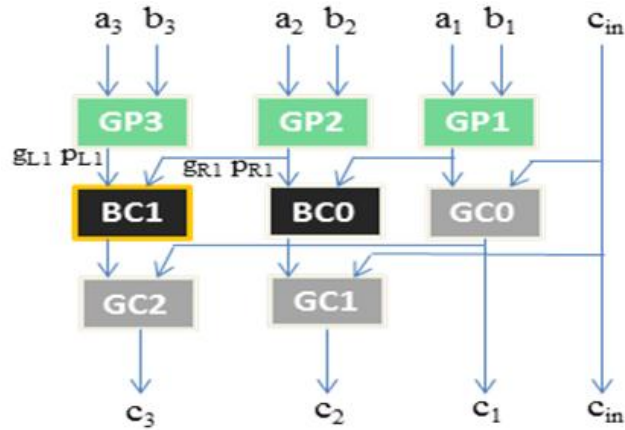


Figure I-4: Kogge stone adder with selected BC1

Case 1: When both the inputs are high:

Case 1(a): Considering BC1

Thus the outputs obtained from the Black Cell 1 (Refer to Figure I-4) when the inputs ‘a₃’ = 1, ‘b₃’ = 1, ‘a₂’ = 1, ‘b₂’ = 1 is shown below in Table I-5

Table I-5: Black cell 1 outputs for high inputs

g _{L1}	p _{L1}	g _{R1}	p _{R1}	g _{L2} = (g _{L1} + p _{L1} × g _{R1})	p _{L2} = (p _{L1} × p _{R1})
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	1

Thus the highlighted case on the Table 1 gives ‘1’ and ‘0’ as the outputs from the black cell 1 when the (g, p) pairs are (1,0) and (1,0). This is the one we are expecting.

Appendix I (Continued)

Case 1(b): Considering GC0

The diagram for the kogge stone adder considering GC0 is as shown in Figure I-5

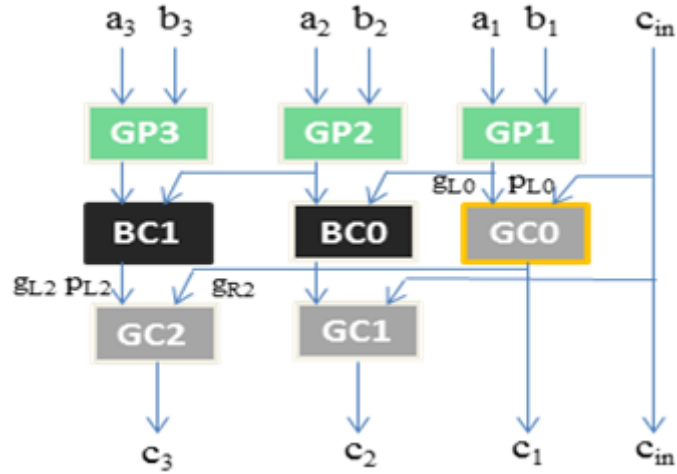


Figure I-5: Kogge-Stone adder with selected GC0

The output obtained from the gray cell GC0 is as shown in Table I-6 when 'a₁' = 1, 'b₁' = 1 and c_{in} = X

Table I-6: Gray cell 0 outputs for high inputs

g_{L0}	p_{L0}	$g_R(c_{in})$	p_R	$g_{R2} = (g_{L0} + p_{L0} \times g_R)$	$(p_{L0} \times p_R)$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	1

Appendix I (Continued)

Case 1(c): Considering GC2

The diagram for the Kogge stone adder considering GC2 is as shown in Figure I-6

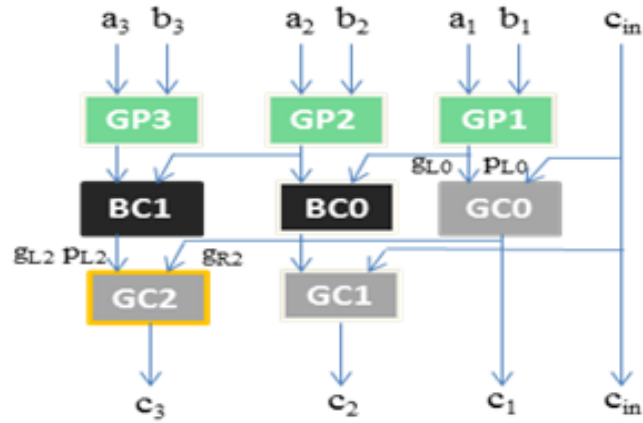


Figure I-6: Kogge-Stone adder with selected GC2

Now the carry out which is shown as ' c_3 ' is calculated as shown in the Table I-7 from the above two cases.

Table I-7: Gray cell 2 outputs for high inputs

g_{L2}	p_{L2}	g_{R2}	p_R	$c_3 = (g_{L2} + p_{L2} \times g_{R2})$	$(p_{L2} \times p_R)$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	1

Thus the carryout for the above input combination when both the inputs are high is '1'

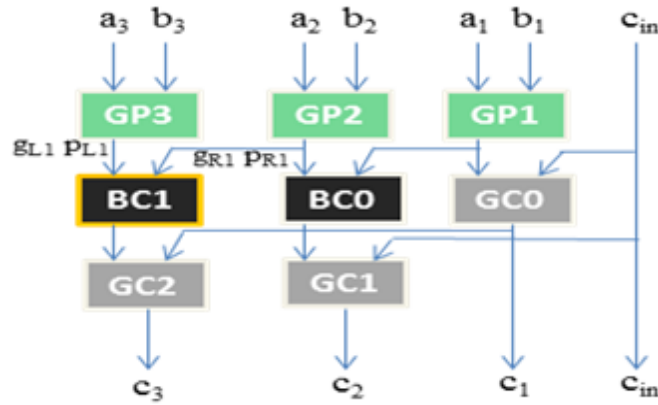
Thus the carryout for case1 is '1'

Appendix I (Continued)

Case 2: When one input is high and one input is low

Case 2(a): Considering Black cell BC1:

When 'a₃' = 0, 'b₃' = 1, 'a₂' = 0, 'b₂' = 1



Thus the outputs obtained from the Black Cell 1 when the inputs 'a₃' = 0, 'b₃' = 1, 'a₂' = 0, 'b₂' = 1 is shown below in Table I-8

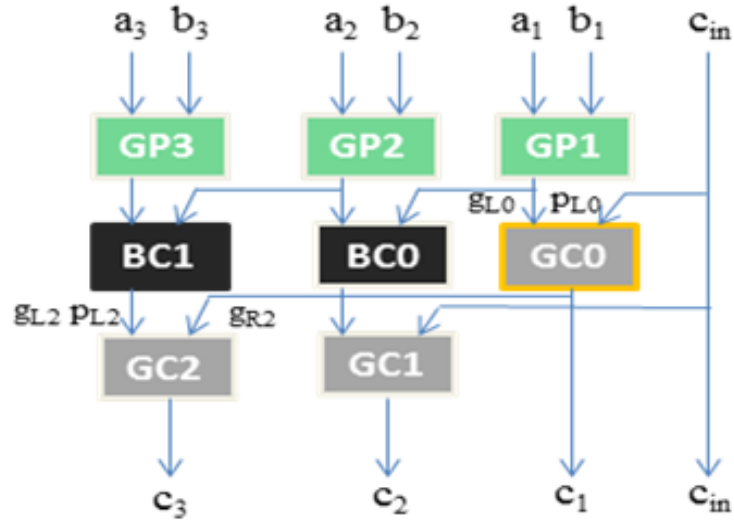
Table I-8: Black cell 1 outputs for one low input

g _{L1}	p _{L1}	g _{R1}	p _{R1}	g _{L2} = (g _{L1} + p _{L1} × g _{R1})	p _{L2} = (p _{L1} × p _{R1})
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	1

Appendix I (Continued)

Thus the highlighted case on the Table I-8 gives ‘0’ and ‘1’ as the outputs from the black cell 1 when the (g, p) pairs are (0,1) and (0,1).

Case 2(b): Considering GC0



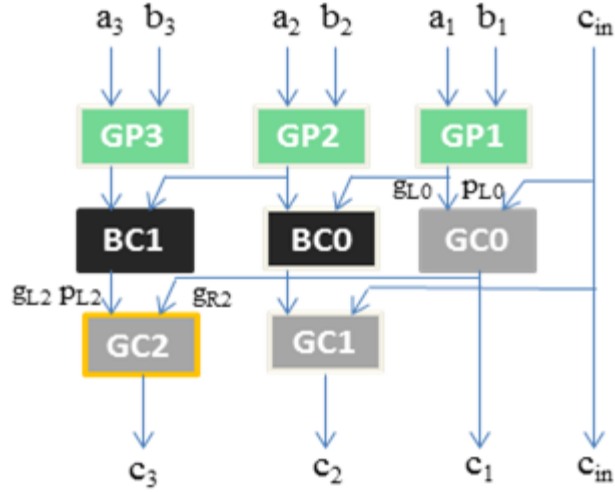
The output obtained from the gray cell GC0 is as shown in Table I-9 when ‘a₁’ = 0, ‘b₁’ = 1 and c_{in} = X

Table I-9: Gray cell 0 outputs for one low input

g _{L0}	p _{L0}	g _R (c _{in})	p _R	g _{R2} = (g _{L0} + p _{L0} × g _R)	(p _{L0} × p _R)
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	1

Appendix I (Continued)

Case 2(c): Considering GC2



Now the carry out which is shown as ' c_3 ' is calculated as shown in the Table I-10 from the above two cases.

Table I-10: Gray cell 2 outputs for one low input

g_{L2}	p_{L2}	g_{R2}	p_R	$c_3 = (g_{L2} + p_{L2} \times g_{R2})$	$(p_{L2} \times p_R)$
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	1	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	1	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	0	1

Appendix I (Continued)

Thus the carryout for the above input combination when one of the inputs are high and other input is low is '0'

Thus carryout in this case2 is '0'.

Observations:

Thus from the above two cases we have observed that the worst case delay will be at the this transition for kogge stone adder

$$\begin{array}{rcl}
 C_{in} = 1 & & C_{in} = 0 \\
 A = FFFF & \Rightarrow & A = 0000 \\
 B = FFFF & & B = FFFF \\
 \hline
 C_{out} = 1 & & C_{out} = 0
 \end{array}$$

For example, the following table represents the input pattern used for obtaining the critical delay of the 16 bit kogge stone adder.

C_{in}	Input A	Input B
1	FFFF	FFFF
0	0000	FFFF
1	FFFF	FFFF
0	0000	FFFF
1	FFFF	FFFF
0	0000	FFFF
1	FFFF	FFFF
0	0000	0000

Note:

The same pattern can also be used for sparse kogge stone adder by following the same procedure discussed above.

Appendix J

Table J-1: Delay Summary for Lower Half Fault Tolerant Sparse Kogge-Stone Adder

16-bit

Corresponding Delay	Gate Delay	Net Delay	Sum Delay
Carry tree	3.06	2.613	5.673
Ripple carry adders	1.693	1.078	2.771
Comparator	3.06	2.495	5.555
Total Delay	13.988 ns		13.988

32-bit

Corresponding Delay	Gate Delay	Net Delay	Sum Delay
Carry tree	3.672	3.228	6.9
Ripple carry adders	1.901	1.078	2.979
Comparator	3.352	2.404	5.754
Total Delay	15.633 ns		15.633

64-bit

Corresponding Delay	Gate Delay	Net Delay	Sum Delay
Carry tree	4.284	3.691	7.975
Ripple carry adders	2.513	1.324	3.837
Comparator	3.404	2.405	5.807
Total Delay	17.621 ns		17.621

128-bit

Corresponding Delay	Gate Delay	Net Delay	Sum Delay
Carry tree	4.896	4.223	9.119
Ripple carry adders	3.501	1.258	4.759
Comparator	3.356	2.599	5.955
Total Delay	19.83 ns		19.833

Appendix J (Continued)

256-bit

Corresponding Delay	Gate Delay	Net Delay	Sum Delay
Carry tree	5.508	4.824	10.332
Ripple carry adders	5.333	1.078	6.411
Comparator	3.2	2.773	5.973
Total Delay	22.716 ns		22.716